LangChain, Microservices, AI, Python, C#

CODE

JAN
FEB
2025

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

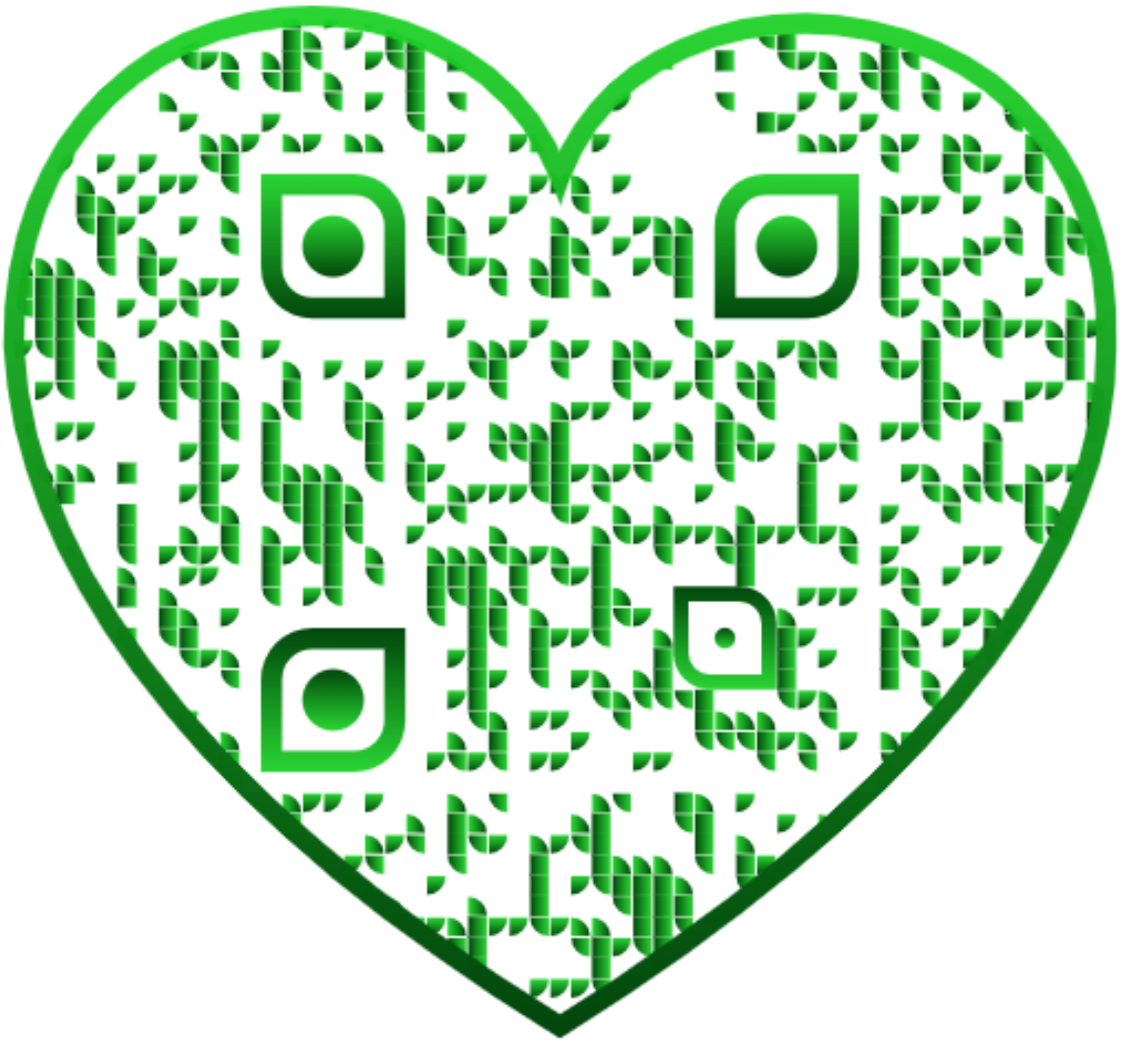
Cover AI generated - Markus Egger

Building AI Pipelines with LangChain

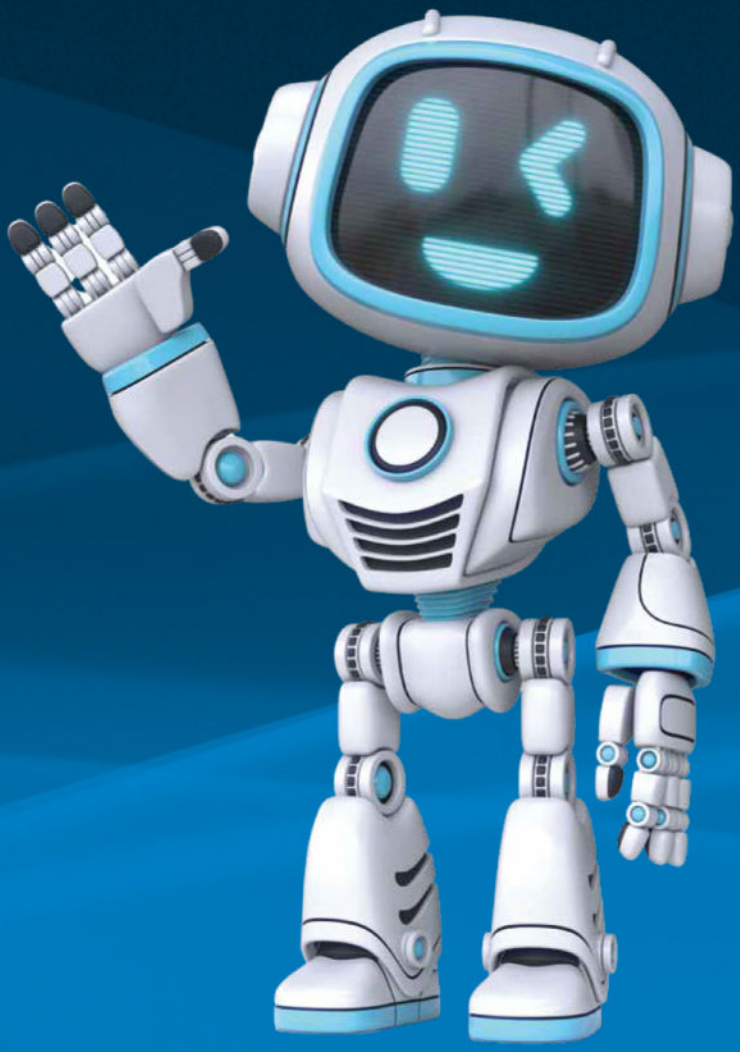
Use CQRS in Microservices

Build Game Encounters with AI

Construct Voice-Enabled AI Applications



CODE
CONSULTING



ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN BENEFIT YOU TODAY?

©shutterstock

EXECUTIVE BRIEFINGS

Are you wondering how AI can help your business? Do you worry about privacy or regulatory issues stopping you from using AI to its fullest? We have the answers! Our Executive Briefings provide guidance and concrete advise that help decision makers move forward in this rapidly changing Age of Artificial Intelligence and Copilots!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

AI-SEARCHABLE KNOWLEDGEBASE AND DOCUMENTS

A great first step into the world of Generative Artificial Intelligence, Large Language Models (LLMs), and GPT is to create an AI that provides your staff or clients access to your institutional knowledge, documentation, and data through an AI-searchable knowledgebase. We can help you implement a first system in a matter of days in a fashion that is secure and individualized to each user. Your data remains yours! Answers provided by the AI are grounded in your own information and is thus correct and applicable.

COPILOTS FOR YOUR OWN APPS

Applications without Copilots are now legacy!

But fear not! We can help you build Copilot features into your applications in a secure and integrated fashion.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/ai-services

832-717-4445 ext. 9 • info@codemag.com

Features

8 Building HAL 9000 (And It Runs Completely on My Mac)

Did "2001: A Space Odyssey" get your interest in AI started early? Come along as Sahil shows you how to make voice activation work for you—offline!

Sahil Malik

16 Exploring .NET MAUI: MVVM, DI, and Commanding

In this fourth entry in his series on MAUI, Paul teaches you about the Model-View-View-Model (MVVM) and Dependency Injection (DI) design patterns to make reusable, maintainable, and testable applications. You'll also learn how to make your code-behind more efficient using Commanding.

Paul Sheriff

28 Building Microservices Architecture Using CQRS and ASP.NET Core

Joydip shows you how the architectural pattern Command Query Responsibility Segregation (CQRS) can help build scalable applications in complex situations.

Joydip Kanjilal

46 Exploring LangChain: A Practical Approach to Language Models and Retrieval-Augmented Generation (RAG)

If you need to use large language models (LLMs) to build applications, you need to know about LangChain, the powerful framework that helps embed LLMs into complex workflows, uses conversational agents, accesses knowledge retrieval systems, and creates automated pipelines. Wei-Meng shows you how to make AI do the work for you.

Wei-Meng Lee

60 Semantic Kernal Part 4: Agents

Following up on his previous articles on core concepts, coding, and advances examples in Semantic Kernal, Mike covers agents, to help you conquer complex tasks and build on their code.

Mike Yeager

64 The Infinite Monster Engine

Even if you've never played it (or gotten hooked), you know about Dungeons and Dragons. You may have envied your friends and co-workers for their lost weekends spent immersed in fantastical worlds. Jason's love of the game is infectious and he shows you how to get AI to help you build some memorable characters using an Infinite Monster Engine.

Jason Murphy

72 My New Copilot+ PC

Mike got a new computer, a Copilot+ PC. "That's nice," you're thinking. Well, wait until he tells you what it can do with Windows Studio Effects, Recall, Phi Silica, Test Recognition, Cocreator with Paint, and Super Resolution! (And what doesn't work yet.)

Mike Yeager

Departments

6 Editorial

11 Advertisers Index

74 Code Compilers

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

• Microsoft Fabric • Databases • Azure Ai •

Microsoft Fabric
COMMUNITY CONFERENCE

March 31 - April 2, 2025 | WORKSHOPS: March 29, 30 & April 3, 2025

MGM GRAND, *Las Vegas, NV*

ARUN ULAGARATCHAGAN
Corporate Vice President, Azure Data, ***Microsoft***

AMIR NETZ
Technical Fellow and CTO of Microsoft Fabric, ***Microsoft***

KIM MANIS
Vice President of Product, Microsoft Fabric & Power BI, ***Microsoft***

ADAM SAXTON
Principal Program Manager, ***Microsoft***

PATRICK LEBLANC
Principal Program Manager, ***Microsoft***

JUSTYNA LUCZNIK
Partner Group Product Manager, Microsoft Fabric Data Engineering & Data Science, ***Microsoft***

PRIYA SATHY
Partner Director, PM, Data Warehousing, ***Microsoft***

SHIREESH THOTA
Corporate Vice President, Azure Databases, ***Microsoft***

WANGUI MCKELVEY
General Manager, Azure Data Analytics, ***Microsoft***

WEE HYONG TOK
Partner Director, Products, ***Microsoft***

Sampling of Optional Pre & Post Conference Workshops

- A Comprehensive Guide to Direct Lake for the Pro Data Modeller (Hands on Lab)
- Build Your First Fabric Workload
- End-to-End Data Engineering: Modern Data Warehousing on Microsoft Fabric
- Fabric Analyst in a Day
- Learn to Develop with Azure AI Services in Microsoft Fabric by Creating DP-700 Exam Study Materials

REGISTER TODAY!

FabricConf.com @MSFabricConf info@FabricConf.com

AI is Stimulating Change

The last two years have brought seismic changes to the entire computing eco system unlike any we've seen before. This tectonic shift has been caused by large-scale research and development efforts in the computing disciplines of artificial intelligence (AI) and machine learning (ML). These technologies aren't new. Amazon has used ML to promote items of interest to you while shopping, your mobile phone completes words as you type with decent accuracy, and customer service bots have been able to handle tasks like returns and order statuses without human intervention.

What's new is the capability, flexibility, and accessibility of these tools to companies and developers that don't have the size or resources of large companies like Apple, Amazon, or Microsoft, etc. Basically, any developer with a small budget and a few simple scripts can begin adding these to their application with minimal effort. This is where we hope CODE Magazine will be a resource for our readers. This issue begins to deliver on this hope.

First, Sahil shows you how to use the content of voice audio passed to an AI model called Whisper, which translates speech into text. The text generated by Whisper is then passed to a large language model (LLM), which answers the questions posed in the speech model.

After that, Wei-Meng Lee does a deep dive into a toolset called LangChain. His article explores how you can create powerful AI pipelines by processing chat requests and refining the results using different models, techniques, and technologies. Not only does the article establish a good foundation for building AI pipelines, but it spends time discussing optimization, memory management, and performance. This is THE article you'll keep next to you when it comes to adding AI features to your application. I know I will.

These articles are joined by several others, including using a new Copilot+ computer, exploring Semantic Kernel, and a really fun one on building role-playing encounters. This issue has a little bit for everyone!

When it comes to AI, the general focus of these pages concentrates on showing you how to use these tools and techniques in your applications. We'll also attempt to provide discussion of the issues surrounding the impact of AI on us as developers, users, and society in general. These technologies are controversial—to say the least—and we'll weigh in where appropriate.

```
WHERE broadcast_week_code IN ({','.join([f'\'{x}\'' for x in set(l_weeks)])})"
```

Figure 1: Some Python code and what VS Code's AI recommended

We started this process some time back in the May/Jun 2023 editorial "The Power of the Pivot" (https://codemag.com/Article/2305011/The-Power-of-the-Pivot). In that editorial, I reviewed a keynote at QCON London where Leslie Miley discussed some of the pitfalls of AI when it comes to power consumption, land use, and equity in general. This talk was given nearly two years ago, and its lessons resonate to this day.

That leads me to a topic that's at the top of my mind. Can we trust the answers given by AI enabled applications? My answer is: PARTIALLY. Or, as the sign says in the pawn shop window, "Buyer Beware." My reason for this answer is that it's been shown that these AI applications can generate incorrect and sometimes "hallucinated" applications. For instance, one of our writers entered his name into a chat application and it turns out he is the co-founder and editor of CODE Magazine. That was definitely news.

In the big picture of things, this is a trivial example. Some AI implementations can cause real issues. Take the article "What Air Canada Lost In 'Remarkable' Lying AI Chatbot Case" from Forbes Magazine (https://www.forbes.com/sites/marisagarcia/2024/02/19/what-air-canada-lost-in-remarkable-lying-ai-chatbot-case/). This article shows how Air Canada lost in small claims court when a bot delivered bad information to a customer. We're seeing article after article with examples of misuse of AI. There are movie trailers that include made-up quotes from real critics, ads with misleading answers to election questions, and the list goes on. It doesn't take long to find info where an AI application has hallucinated some information that seems to apply—on the surface.

Given all this, why did I give AI a trust level of PARTIALLY? It's because for some uses, these AI models are just good enough, provided that some human reviews their output for accuracy.

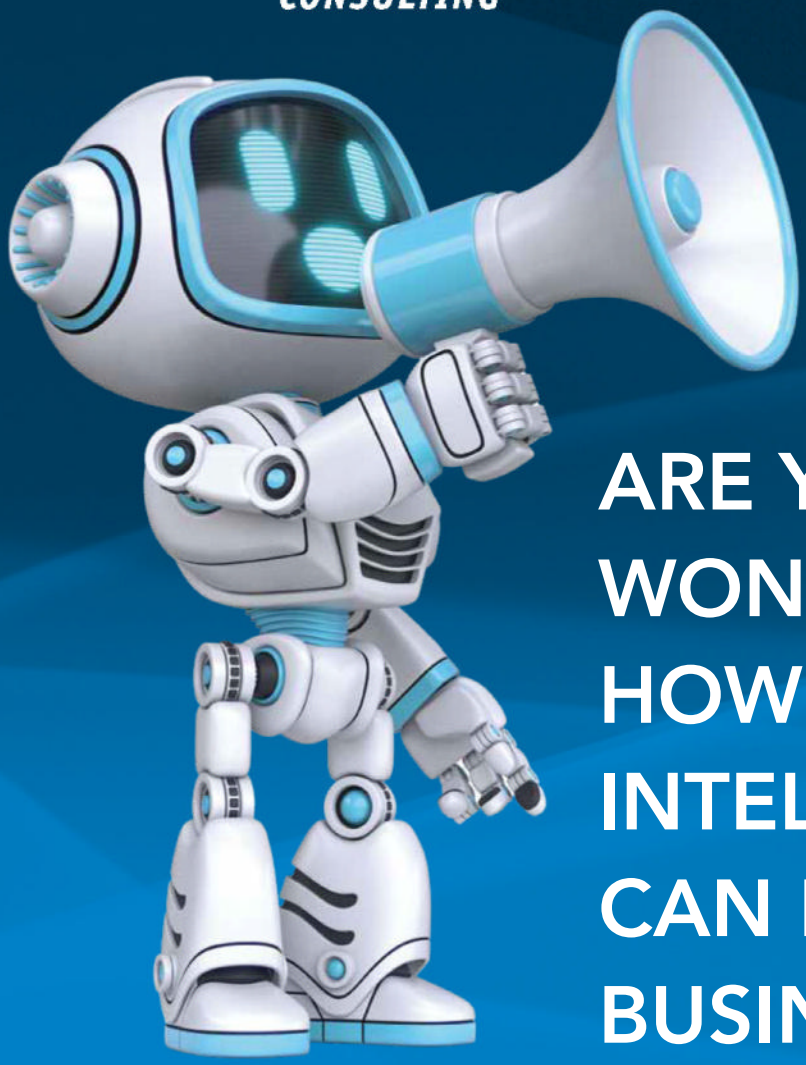
As an example, **Figure 1** shows some Python code I was working on in VS Code.

The grey part of the code is what was recommended by VS Code immediately after I hit the space bar after the last character I entered. Should I rely on this code without consideration and examination? Absolutely not! It's up to me to validate the accuracy and suitability of the code I write. How did it do? Not too bad, in this humble developer's opinion. Helping coders be more productive is where these AI applications shine. Programming has many well-known concepts and techniques, and these AI tools do a fairly good job of incorporating them into developer tools. When it comes to output from these AI tools, I follow the old Reagan concept of "Trust but Verify."

What really matters in the usage of these applications is suitability for the job. As a developer, you need to determine whether usage of AI has non-trivial risk associated with its responses. This is where we will be facing challenges for some time to come. In the pages of CODE Magazine, we'll try to raise these questions and discuss potential answers to them. Now it's time to use AI to build an encounter for that skeletal dragon mini I'm painting!

Rod Paddock

CODE CONSULTING



ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN HELP YOUR BUSINESS?

Do you worry about privacy or regulatory issues stopping you from using AI to its fullest?

We have the answers!

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

codemag.com/executivebriefing

832-717-4445 ext. 9 • info@codemag.com

©shutterstock

Building HAL 9000 (And It Runs Completely on My Mac)

HAL 9000 is a fictional artificial intelligence (AI) character in Arthur C. Clarke's novel "2001: A Space Odyssey" and its film adaptation directed by Stanley Kubrick. It had a conversational interface—humans could just talk to it like humans talk with each other. It was super intelligent. The original idea came about in 1964 when Kubrick and Clarke started working on the project.

Sahil Malik
www.winsmarts.com
@sahilmalik

Sahil Malik is a Microsoft MVP, INETA speaker, a .NET author, consultant, and trainer.

Sahil loves interacting with fellow geeks in real time. His talks and trainings are full of humor and practical nuggets.

His areas of expertise are cross-platform Mobile app development, Microsoft anything, and security and identity.

The year 1964 was when the Ford Mustang was introduced. Rotary phones were how you made phone calls, and you wore thick black polycarbonate glasses. Microwave ovens were a new thing, as were color TVs. As the world danced to "Pretty Woman" by Roy Orbison, or "Twist and Shout" by the Beatles, society was looking forward to the seemingly impossible goal of putting man on the moon by the end of the decade.

It's in that enchanting time that HAL 9000 was imagined, a super intelligent computer program that could control the functions of a spaceship, that was self-aware, and people could interact with in natural language.

Fast forward to 2024. Although I don't quite yet have my personal flying spaceship, HAL 9000 is a pretty close reality. Nerd that I am, I set out to build HAL 9000 for myself.

In my last article in CODE Magazine, I talked about running AI locally. The goal of this article is different but straightforward. I want to build a HAL 9000 that I can speak to, in any language, about any topic. And it should give me answers about whatever I ask for. Additionally, I want to do it all on my off-the-shelf commercially available MacBook Pro. Finally, I want to be able to build it so that it runs completely offline, so that in the rare case I manage to get a spaceship, I don't have to rely on an internet connection to run it.

As I build it, I'll share all of the code and explain it as I go. In the end, I'll put together a fully functioning application, HAL 9000.

What You're Going to Need

To follow this article, you'll need a beefy machine. You're not going to rely on the cloud to build the model for you. You'll need a powerful local compute capability. This means either a higher-end Windows/Linux laptop or one of the newer Macs. And yes, you'll need a GPU. AI involves a lot of calculations and to speed things up, a lot of them are offloaded to the GPU. The difference between doing everything on the CPU vs. GPU is astronomical. For my purposes, I'll use my rusty trusted M1 Max MacBook Pro. It's a few years old, but it has enough oomph to work on thousands of pages of text, which is good enough for my needs. Hopefully, you have a similarly equipped machine, or, to follow along you could just use a smaller input dataset.

Also, you'll download and use standard libraries, packages, and large language models that other companies and people have built. But when you're done with it, no data will be sent to the cloud, and your application will have the ability to be able to run completely offline. To get started, though, you'll need an internet connection.

Also, I will use Python, so ensure that you have Python 3x installed.

The Main Components

Let's think about the problem and break it down into smaller parts.

I'll need the ability to listen and convert my spoken text into ASCII text. When I speak into my mic, saying "Let's talk about New York City," my program should be able to transcribe the text I say on the fly.

I'll need a large language model (LLM) that takes my spoken text, transcribed to plain text, as inputs, and returns a sensible response.

And finally, I'll need the ability to take the LLM's response and convert it to audio, which I can then hear through my speakers.

All this put together should give me a super intelligent sentient being.

Enter Hugging Face

Hugging Face (http://huggingface.co) is a popular open-source AI community and platform focused on natural language processing (NLP) and transformer-based models. It has a pretty impressive transformers library, a number of pre-trained models, a model hub where you can find or contribute to models, a large number of datasets for your own experimentation, and so much more. I figured that this would be a great place for me to start exploring what I can build.

For my purposes, models are what I care about. I went to https://huggingface.co/models, started looking at the various models, and found that there are a number of impressive models available. As I started playing with them, I started discovering superpowers. For instance, a long time ago, I saw that Microsoft Cognitive services (now known as Microsoft AI services) had a REST API that you could just show a picture to and it would detect what it saw in the picture. Or draw bounding boxes around things it saw in the picture. There's a whole section of models dedicated to computer vision. I tried a few capabilities there, like I showed it a picture of my dogs, and it immediately recognized the dogs in the picture, even the

breeds; or I could have a conversation about the image, like "Tell me more about a Doberman" etc. Theoretically speaking, I could just show my AI bot a picture from my webcam, and start talking about it. Or I could run it on my phone and show it a weed, have it recognize the plant, and then tell me how to get rid of it. Or how about a bunch of scanned receipts and ask something like, "How much did I spend on Burritos last year?" I'll leave computer vision for another day. For now, let's re-focus on audio.

There are a bunch of models available under the Audio section of the models page in Hugging Face. There are text-to-speech models, text-to-audio, automatic speech recognition, audio-to-audio, audio classification, and voice activity detection.

For my needs, I already know that I will find text-to-speech and automatic speech recognition useful.

Automatic Speech Recognition

This is the first problem I need to solve. I need to be able to speak to the computer, hopefully in any language, and it should be able to transcribe the texts with decent accuracy. I noticed that one of the models available was openai/whisper. So I decided to play with it.

Whisper (https://github.com/openai/whisper) is a general-purpose speech recognition model. It's trained on a large dataset of diverse audio and is also a multitasking model that can perform multilingual speech recognition, speech translation, and language identification. In order to use Whisper on my Mac, I needed to install FFmpeg first. Well, that's easy. I was able to install that using the command below.

```
brew install ffmpeg
```

With the above in place, I started writing my first simple Whisper-based program. The first step is to define a Python .venv, which I'll skip the details of since I assume you're already familiar with the basics of Python.

With that in place, let's define the requirements.txt, which is shown below

```
soundfile
pyaudio
SpeechRecognition
git+https://github.com/openai/whisper.git
```

I defined a launch.json in my .vscode folder that allowed for debugging, which can be seen in **Listing 1**. And now my canvas was set up to start playing with Whisper. I went ahead and wrote the following simple program:

```
import whisper

model = whisper.load_model("base")
result = model.transcribe("audio.mp3")
print(result["text"])
```

To my shock, whatever text was spoken in audio.mp3, Whisper transcribed accurately into text. I do notice that these open-source models show a lot of errors and warnings. Those errors and warnings are worth paying heed to. For the purposes of this article, I want to keep my outputs clean, so I suppressed them, as follows.

Listing 1: My launch.json to enable Python debugging

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python Debugger: Current File",
            "type": "debugpy",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal"
        }
    ]
}
```

Listing 2: Live audio transcribing using Whisper

```
import speech_recognition as sr
from speech_recognition import
  Microphone, Recognizer, UnknownValueError

r = sr.Recognizer()
with sr.Microphone(device_index=7) as source:
    print("Say something!")
    audio = r.listen(source)

try:
    print("You said:" +
        r.recognize_whisper(audio,
          language="english"))
except sr.UnknownValueError:
    print("Didn't understand")
except sr.RequestError as e:
    print(f"Could not request results; {e}")
```

```
import logging
import warnings

warnings.filterwarnings('ignore')
for name in
    logging.Logger.manager.loggerDict.keys():
     logging.getLogger(name).setLevel(
       logging.CRITICAL)
```

Okay, I'm excited. My simple audio transcription is working. I also want to be able to talk into a microphone and have Whisper transcribe it on the fly. I played around a bit with Whisper and was able to write code, as shown in **Listing 2**. This code allows me to speak into the microphone, and Whisper can detect it. A curious thing you see in **Listing 2** is "device_index=7". What is that magic number 7? Well, 7 isn't just my lucky number; it's the index of the microphone I wish to listen to. To list all microphones on your computer, just use the below code snippet:

```
for index, name in
    enumerate(
      sr.Microphone.list_microphone_names()):
    print("Microphone with name \"{1}\"
       found for `Microphone(device_index={0})`".
       format(index, name))
```

Now let's run my code and see how it works. To run the code in VSCode, I simply hit F5, and once the code prints "Say something," I just said whatever I wished. The output can be seen in **Figure 1**.

As exciting as this is, I couldn't help but wonder: What if I wanted to make this smarter? As in, be able to speak in any language, detect the language, and use that detected language to both chat with my LLM and use it for audio transcription. This can be done using the code shown in **Listing 3**.

```
✘ sahilmalik@MakhiMax ~/Documents/Developer/cm/article2/audio cd /Users/sah
er/cm/article2/audio ; /usr/bin/env /Users/sahilmalik/Documents/Developer/cm/arti
on /Users/sahilmalik/.vscode/extensions/ms-python.debugpy-2024.10.0-darwin-arm64/
ter/../../debugpy/launcher 58083 -- /Users/sahilmalik/Documents/Developer/cm/arti
udio.py
Say something!
You said: Hello, how are you doing today?
sahilmalik@MakhiMax ~/Documents/Developer/cm/article2/audio
```

Figure 1: Speech recognition seems to work.

```
✘ sahilmalik@MakhiMax ~/Documents/Developer/cm/article2/audio
er/cm/article2/audio ; /usr/bin/env /Users/sahilmalik/Documents/Deve
on /Users/sahilmalik/.vscode/extensions/ms-python.debugpy-2024.10.0-
ter/../../debugpy/launcher 58169 -- /Users/sahilmalik/Documents/Deve
udioContinously.py

 I am listening, start talking..

 Are you listening?
 I'm glad you're listening. What are we going to talk about today?
 I'm impressed by the speech recognition quality, but how well does
 This is quite impressive, but it isn't doing anything yet other tha
 We need to make it smarter.
 Goodbye
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/mu
 UserWarning: resource_tracker: There appear to be 1 leaked semaphor
  warnings.warn('resource_tracker: There appear to be %d '
sahilmalik@MakhiMax ~/Documents/Developer/cm/article2/audio
```

Figure 2: Output of continuously recognizing spoken text

Listing 3: Audio transcribing in any language

```
import whisper

model = whisper.load_model("base")

audio = whisper.load_audio("audio.mp3")
audio = whisper.pad_or_trim(audio)
mel = whisper.log_mel_spectrogram(audio).to(model.device)
_, probs = model.detect_language(mel)
print(f"Detected language: {max(probs, key=probs.get)}")

options = whisper.DecodingOptions()
result = whisper.decode(model, mel, options)

print(result.text)
```

This is incredible, but I'm going to keep things simple by limiting this article to English. You can imagine how easy it would be to modify this bot to work in any language by just detecting the language being spoken and passing that as an input to the recognized language to converse in.

As impressive as this is, I want my audio transcribing to work continuously. In other words, until I say a catch phrase like "Goodbye," I want my program to keep doing audio transcriptions. After all, as I'm writing the chat bot, I'm going to have a conversation with it. I'll say things like, "Let's talk about New York City," and it'll tell me some general information about New York City, and then I might ask further questions based on the context of the answer.

The code to listen for audio continuously until I say "Goodbye" can be seen in **Listing 4**. This code running in my VSCode's debug output can be seen in **Figure 2**.

One thing I'll say about the output you see in **Figure 2**, is that although the transcription is shockingly accurate, and works across multiple languages, I did notice that any background noise can easily confuse it. So if you're following me along in actual code, try and do this in a quiet environment. That said, there are tweaks you can make to tune out background noise.

All right, I think we have the first ingredient of the bot all done. Let's make it smarter by connecting my spoken text to a large language model.

Connecting My Text to a large LLM

A large language model (LLM) is a type of artificial intelligence (AI) designed to process and understand human language, typically using deep learning techniques. There are many large language models, specialized for various needs. You could pick from any number of large language models available on Hugging Face. All the big popular names like Llama and Gemma and Phi, etc., are available.

Now, large tech companies have spent billions of dollars building these models, so they don't just give it away. For most models, you'll have to fill out a form and acknowledge terms of use, and, for some models like Llama3.1, you must wait for approval. In certain jurisdictions, they may not allow you to use the model at all.

Listing 4: Listen for spoken text continuously

```
import os
import speech_recognition as sr
from speech_recognition import
   Microphone, Recognizer, UnknownValueError

def audio_callback(recognizer, audio):
    try:
        prompt = recognizer.recognize_whisper(
           audio, model="base", language="english")
        print(prompt)
        if "bye" in prompt.lower():
            stop_listening(wait_for_stop=False)
            os._exit(0)
    except UnknownValueError:
        print("There was an error processing the audio.")

recognizer = Recognizer()
microphone = Microphone(device_index=7)

with microphone as source:
    recognizer.adjust_for_ambient_noise(source)

stop_listening =
  recognizer.listen_in_background(microphone,
    audio_callback)
input() # wait to exit
stop_listening(wait_for_stop=False)
```

You could take any of these models, and fine tune them also. Fine-tuning a Large Language Model (LLM) involves adjusting the model's weights and parameters to better perform a specific task or adapt to a particular domain. By fine tuning, you can improve performance for a specific task, you could adopt to your specific domain, you can reduce bias, or you might improve model generalizability, as need be. I'll leave fine tuning for a future article. For now, I feel that a generic LLM will suffice.

I started playing around with a few models, and decided I'll use Gemma for this article. Gemma (Generative Expert Memory Model Architecture) is an AI model developed by Google. Google has put in all the hard work already. They've trained it on 45 terabytes of data, and it's incredible and knowledgeable in so many fields. I found Gemma to be great for conversational AI, and with minimal prompt engineering, I was able to get it to give me coherent answers that were fun and useful. I'm not saying Llama is bad; to be honest, all of these models are quite comparable to each other.

Let's start building the app with Gemma. Because I'm running everything locally, I went with the 2-billion parameter version of Gemma. The more parameters, the better your accuracy, but the more beefy computer you're going to need to run this. How about using LangChain to do something smart? When you have access to puny hardware, run the 2-billion parameter model, and when you have a server, call out to the server running the 45B parameter. When you're online, call out to OpenAI and some model there.

To use this model, I had to visit https://huggingface.co/google/gemma-2-2b-it and on the right hand top, click on the "Use this model" button and select "Using Transformers". Right there it showed me some example code. Well, this is too easy.

I must say that you'll have to frequently fill out an acknowledgement form, and the team building Llama must allow you to use a model. This is as simple as going to the page for Llama on Hugging Face, filling out a simple form, going to your Hugging Face account settings, and creating an access token for yourself at https://huggingface.co/settings/tokens. With all that in place, just include the below snippet in your code and you're good to go.

```
from huggingface_hub import login
login("yourtoken")
```

Now back to Gemma, where you don't need to have an authentication token, let's start writing code for the conversational bot. The first step is to create a pipeline object, as can be seen below.

```
import torch
from transformers import pipeline

pipe = pipeline(
    "text-generation",
    model="google/gemma-2-2b-it",
    model_kwargs={"torch_dtype": torch.bfloat16},
    device="mps",
)
```

The "mps" is because I'm running on a Mac. If you're on a PC with an NVIDIA card, just replace that with "cuda".

A transformer pipeline in AI refers to a sequence of processing stages that use transformer architectures to perform specific tasks. At the bare minimum, a pipeline takes a model to make predictions from inputs, and

Listing 5: An example of a pipeline for sentiment analysis

```
import pandas as pd
from transformers import
  pipeline, AutoTokenizer,
    AutoModelForSequenceClassification

# Load pre-trained model and tokenizer
model_name =
  "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model =
   AutoModelForSequenceClassification.from_pretrained(
 model_name)

# Create pipeline
classifier = pipeline("sentiment-analysis",
  model=model, tokenizer=tokenizer)

# Example text
text = "I loved the new movie!"

# Run pipeline
result = classifier(text)

print(result)
```


ADVERTISERS INDEX

Advertisers Index

CODE Consulting--AI Services
www.codemag.com/ai-services 2

CODE Consulting--Executive Briefing
www.codemag.com/executivebriefing 7

CODE Consulting--Copilot
www.codemag.com/copilot 59

CODE Consulting
www.codemag.com/Code 75

CODE Staffing
www.codemag.com/staffing 76

dtSearch
www.dtSearch.com 15

Every Woman Tech
www.everywoman.com/tech-forum 35

Geo Week
www.geo-week.com 45

Microsoft Fabric Community Conference
www.fabricConf.com 5

Advertising Sales:
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

This listing is provided as a courtesy to our readers and advertisers. The publisher assumes no responsibility for errors or omissions.

Listing 6: My fancy text based chat bot

```
import torch
from transformers import pipeline

pipe = pipeline(
    "text-generation",
    model="google/gemma-2-2b-it",
    model_kwargs={"torch_dtype": torch.bfloat16},
    device="mps",
)

def generate_text(prompt, previousResponses):
    prompt = prompt + ". Answer in brief."
    allPrevResponses = ""
    for previousResponse in previousResponses:
        allPrevResponses += previousResponse + "\n"
    messages = [
        {"role": "user", "content": allPrevResponses +
 "\n" + prompt},
    ]
    outputs = pipe(messages, max_new_tokens=256)
    assistant_response =
 outputs[0]["generated_text"][-1]["content"].strip()
    return assistant_response

previousResponses = []
while True:
    user_input = input("\033[92m >> You: \033[0m")
    response = generate_text(user_input, previousResponses)
    previousResponses.append(response)
    print("\033[93m >> AI:", response, "\033[0m")
```

a tokenizer for mapping raw text inputs to a token. A tokenizer is simply a component that splits text into individual words, phrases, or sub words, called tokens. For instance, you may have something like, "take some text and do sentiment analysis on it," and you'd need a model to perform this task. Using that model, you'd create a pipeline. Although not the focus of this article, I wanted to show you how simple it is to build sentiment analysis. You can see the code for sentiment analysis using a popular model in **Listing 5**. As you can see, it's a matter of having a model and a tokenizer, building a pipeline and using it.

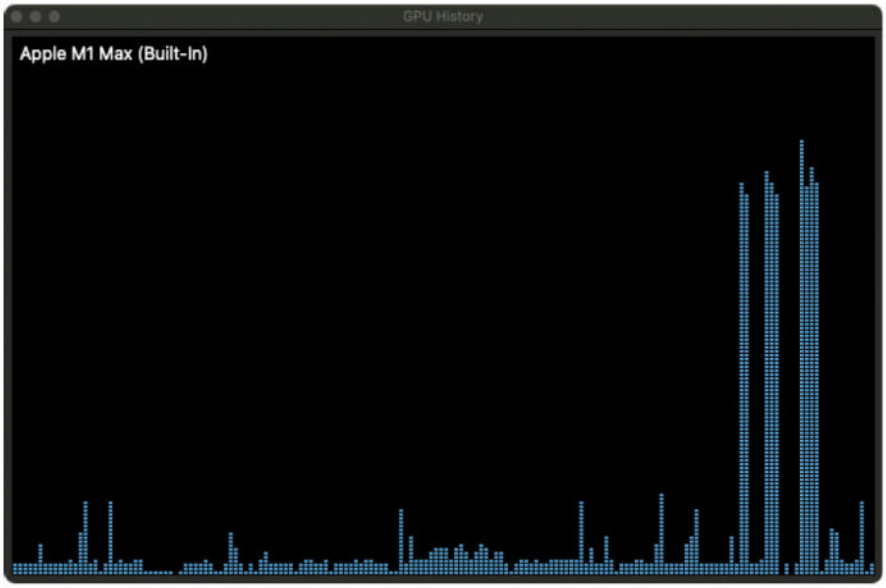


Figure 4: My hardworking GPU

It would be cool to pair this sentiment analysis with the simple old bot and make it smart enough that if the conversation is becoming too sad, it'll throw a joke in there for fun. I'll leave that as an experiment for you to do.

But let's refocus on my intelligent bot and get back to building it. With my pipeline set up, I can give it an input as "prompt" and the large language model returns me the output I'm looking for. This is as simple as the code snippet you see below.

```
messages = [
{"role": "user", "content": prompt},
]
outputs = pipe(messages, max_new_tokens=256)
assistant_response =
  outputs[0]
    ["generated_text"][-1]["content"].strip()
```

Feel free to run this, and you'll be able to get an answer to any question you may have. But that's not what we're

```
 >> You: Lets talk about new york city
 >> AI: New York City: a global hub of culture, finance, and energy. 🏙️ Known for its iconic skyline, dive
rse neighborhoods, world-class museums, and vibrant street life.  What specifically interests you about NY
C? 🗽🍎
 >> You: How far is New york city from Washington DC? Answer in pirate style.
 >> AI: Avast ye, matey! NYC's got it all: the sights, the sounds, the food!  A city that never sleeps, wi
th a heart full o' dreams and a soul full o' grit. 🗽🍎

As for the distance, it be a mere 180 nautical miles, or a short sail across the Hudson! 🏴‍☠️
 >> You: What can I do there?
 >> AI: You've got a lot of options in NYC!

**Must-Dos:**

* **Iconic Landmarks:**  Statue of Liberty, Empire State Building, Times Square
* **Museums:** Metropolitan Museum of Art, Museum of Modern Art (MoMA), American Museum of Natural History
* **Neighborhood Exploration:**  Walk through Greenwich Village, explore Chinatown, wander through Brookly
n's hipster havens
* **Food Scene:**  Grab a slice of pizza, indulge in a bagel, try a street vendor's hot dog

**Don't forget:**  Catch a Broadway show, enjoy a rooftop bar, and soak up the city's energy!
 >> You:
```

Figure 3: Me talking with my chatbot

Listing 7: My audio driven chat bot

```
import os
import speech_recognition as sr
from speech_recognition
  import Microphone, Recognizer, UnknownValueError
import torch

import torch
from transformers import pipeline

import logging
import warnings

warnings.filterwarnings('ignore')
for name in logging.Logger.manager.loggerDict.keys():
    logging.getLogger(name).setLevel(logging.CRITICAL)

pipe = pipeline(
    "text-generation",
    model="google/gemma-2-2b-it",
    model_kwargs={"torch_dtype": torch.bfloat16},
    device="mps",
)
def AskAI(prompt, previousResponses):
    prompt = prompt + ". Answer in brief."
    allPrevResponses = ""
    for previousResponse in previousResponses:
        allPrevResponses += previousResponse + "\n"

    messages = [
        {"role": "user",
       "content": allPrevResponses + "\n" + prompt},
    ]
    outputs = pipe(messages, max_new_tokens=256)
    assistant_response =
      outputs[0]["generated_text"][-1]["content"].strip()
    return assistant_response

previousResponses = []

def audio_callback(recognizer, audio):
    try:
        prompt = recognizer.recognize_whisper(
          audio, model="base", language="english")
        print("\033[92m >> You: " + prompt + " \033[0m")
        print("\r Thinking ")
        response = AskAI(prompt, previousResponses)
        previousResponses.append(response)
        print("\r\033[93m >> AI:", response, "\033[0m\n")

        if "bye" in prompt.lower():
            stop_listening(wait_for_stop=False)
            os._exit(0)
    except UnknownValueError:
        print("There was an error processing the audio.")

recognizer = Recognizer()
microphone = Microphone(device_index=7)

with microphone as source:
    recognizer.adjust_for_ambient_noise(source)

stop_listening =
  recognizer.listen_in_background(
   microphone, audio_callback)

print("\n ----------------------------- \n
  I am your super friendly AI,
  what do you wanna chat about today? \n ")
input() # wait to exit
stop_listening(wait_for_stop=False)
```

trying to do. We want our bot to be smarter. We want it to understand context. So if I say, "Let's talk about dogs" with a subsequent question about "What is bark?", I want to know about dogs barking. But if I say, "Let's talk about trees" with a subsequent question of "What is bark?", I wish to be told about tree bark.

Context is important. For example, "my dogs love to play in leaves" or "my dogs are not happy when their owner leaves", have two entirely different meanings for the same word.

There are two ways to attach context. One is that instead of **role : user**, you can just attach all previously said text as **role : assistant**. Alternatively, the prompt can just remember all previously generated text. I'll use the latter approach and put together a full code example, as can be seen in **Listing 6.** Notice that in **Listing 6** I have also added a prompt of "Answer in brief." I found Gemma to be a bit too wordy. Or maybe I'm just impatient.

My interaction with Gemma can be seen in **Figure 3**. Remember, this is me typing into a keyboard.

Notice that all the processing is being done locally; look at how hard my GPU is working in **Figure 4** when I ask it all these questions.

There are some other interesting things you can see in **Figure 3**. My second question said, "Answer in pirate style" and the model did. This is so funny. But in my next question, I just said "What can I do there?" But what is "there"? My model understood from the context that I'm still talking about New York City.

```
 >> AI: ## Cooking 101: A Quick Start

**1. Safety First:**
   - Always wash your hands and surfaces.
   - Use sharp knives safely.
   - Be careful with hot oil and stovetop.

**2. Basic Techniques:**
   - **Sauté:** Quickly cook food in a pan with oil over medium heat.
   - **Roast:** Cook food in an oven at a high temperature.
   - **Simmer:** Cook food in liquid at a low temperature.
   - **Steam:** Cook food in a steamer basket over boiling water.

**3. Essential Tools:**
   - Chef's knife
   - Cutting board
   - Measuring cups and spoons
   - Pots and pans
   - Spatula
   - Whisk

**4. Start Simple:**
   - **Eggs:** Scrambled, fried, poached.
   - **Pasta:** Boil, toss with sauce.
   - **Roasted Vegetables:** Season and roast in the oven.
   - **Grilled Cheese:** A classic comfort food.

**5. Resources:**
   - **Cookbooks:** Find recipes for your skill level.
   - **Online:** Websites and videos offer countless cooking tutorials.
```

Figure 5: AI helping me with cooking

```
 >> AI: 1. **Butter** one side of each bread slice.
2. **Place** bread slices in a pan over medium heat.
3. **Add** cheese to one slice.
4. **Top** with the other slice of bread.
5. **Cook** until golden brown and cheese is melted.
```

Figure 6: AI helping me with cooking

Listing 8: Text to audio

```
import sounddevice as sd
import ChatTTS

chat = ChatTTS.Chat()
chat.load(compile=True)

texts = [
    "how are you?"
    ]

params_infer_code =
  ChatTTS.Chat.InferCodeParams(
    temperature = .3,
    top_P = 0.7,
    top_K = 20,
)
wavs = chat.infer(texts,
  params_infer_code=params_infer_code)

sd.play(wavs[0][0], 24000, blocking=True)
```


SPONSORED SIDEBAR

Stay Connected with CODE Magazine on LinkedIn

Join our professional community on LinkedIn to stay ahead in the tech world.

Exclusive Content: Get the latest tech insights and articles.

Industry Updates: Stay informed with cutting-edge news.

Networking: Connect with peers and industry leaders.

Help us grow by sharing with colleagues and friends.

Follow us today: www.linkedin.com/company/code-magazine

If you're curious, I played the "Let's talk about dogs" and "Let's talk about trees" game and asked "Tell me about bark." Here are the outputs I received:

> Bark is a dog's way of communicating, like a "hello" or a "warning." It's a complex sound with many meanings depending on the context and the dog's tone.
>
> Bark is fascinating! It's the tree's protective outer layer, a shield against insects, disease, and the elements. It also plays a role in water and nutrient transport, and can even change color and texture with age.

Putting It All Together

I think we have a pretty impressive application in the works. Let's put all this together now. I can convert text to audio, I can listen continuously. I can converse with an LLM. I can chat based on context. Putting **Listing 4** and **Listing 6** together, I get **Listing 7**, which is my fully functional chatbot that I can speak with.

Let's give it a try. If you're online, I put together a few examples of this program running.

You can see me chat about Microsoft Graph here: https://www.youtube.com/watch?v=8vJtldKwxKw. Or you can see me chatting about airplanes here: https://www.youtube.com/watch?v=rnn9hLdvWv4. Or you can see me learn about securing JavaScript applications here: https://www.youtube.com/watch?v=0GnwGJgiABQ.

It's probably more compelling to watch a video and hear me talk and get a feel for how this works interactively than to see it in text, but if you're not online, let's see some fun interaction here.

Let's say I'm throwing a party and I've never cooked anything. My first question to AI is:

> Teach me how to cook.

And it gives me a nice, detailed output, as can be seen in **Figure 5**.

I like the "Start simple" bit, so let me ask how to make a grilled cheese.

> How do I make a grilled cheese?

As expected, AI gives me quick five steps to make a grilled cheese. This can be seen in **Figure 6**.

Now, that I've established a context, I can just ask a subsequent question.

> Will it make me fat?

To which AI promptly replies,:

> No, a grilled cheese is unlikely to make you fat if you eat it in moderation as part of a balanced diet.

Well, that's good to know! Feel free to keep this conversation going about any other topic you wish.

Text to Audio

All this chat about grilled cheese is making me hungry, so let me leave you with a little teaser. You've so far built a fully functional chatbot. I've played a bit with it: I asked it about Microsoft Graph, programming in general, security-related stuff, I asked it about cooking as you saw, and I asked it about touristy stuff. I asked it about investing. I asked it about historical events. In every instance, my jaw was on the floor.

I did talk about a few extensibility points, like detecting language, and adding sentiment analysis. But to round up the chatbot, let's add the last bit, which is text to audio.

This is where the real world kicks in. In the movie, HAL answered questions that were suited to short audio interactions. That isn't how the real world operates. Sometimes the output is code. Sometimes it's images. Other times it's bulleted lists. Just look at **Figure 5**. Now imagine closing your eyes and having that text read out to you as audio. I wouldn't find that very useful, to be frank. It's easier to read bulleted lists yourself than to have them read out to you.

Still, because I set out to do a full audio-based interaction, let's add text to speech also.

Back on Hugging Face, I found the **2noise/chatts** model as the most popular text-to-audio model. It was easy to put together a code example that converted any input text to a pretty decent quality of spoken audio. You can see the code for text-to-audio in **Listing 8**. In fact, I was able to visit https://chattts.com and tweak the inputs to figure out what parameters worked best for me.

I'll leave it up to you to integrate it into **Listing 7**, but as I said, I didn't find it very useful except for simple questions that had straightforward and to-the-point answers. For example, quick math questions, or asking factual questions that didn't need long drawn-out answers.

Summary

Did I just build a super intelligent chat bot that I can talk with about any topic and it replies intelligently? Yeah, I just did. And I've been using it to learn about all sorts of stuff. I've heard stuff like the federal reserve is lowering rates and it's taught me all about it.

I find it simply amazing that what was purely science fiction in 1964, I was able to effectively build over the weekend. Just to drive home the point of how incredible this is, what else was considered science fiction in 1964? What's considered science fiction today? Teleportation? Cloning? Travel at the speed of light? Time travel? Admittedly, 1964 was 60 years ago, but that's still within one person's lifetime.

Can you imagine, in the year 2094, some random guy using off-the-shelf hardware to clone himself, time travel back to 2024, and write an article to show how to build HAL?

Yeah, neither can I.

Well, I'm off to make a grilled cheese. You have fun.

Sahil Malik

dtSearch®

Instantly Search Terabytes

dtSearch's **document filters** support:
- popular file types
- emails with multilevel attachments
- a wide variety of databases
- web data

Over 25 search options including:
- efficient multithreaded search
- easy multicolor hit-highlighting
- forensics options like credit card search

Developers:
- SDKs for Windows, Linux, macOS
- Cross-platform APIs cover C++, Java and current .NET
- FAQs on faceted search, granular data classification, Azure, AWS and more

Visit dtSearch.com for
- hundreds of reviews and case studies
- fully-functional enterprise and developer evaluations

The Smart Choice for Text Retrieval® since 1991

dtSearch.com 1-800-IT-FINDS

Exploring .NET MAUI: MVVM, DI, and Commanding

Up to this point in this article series on .NET MAUI ,you created a set of typical business application input pages and learned about the many different controls you can use for data input. Data binding is a great feature of .NET MAUI to help you eliminate C# code in your applications. In this article, you'll learn the Model-View-View-Model (MVVM) and Dependency Injection (DI) design patterns to create reusable, maintainable, and testable applications. You'll learn to eliminate code in your code-behind by taking advantage of Commanding. You'll also learn how to apply Commanding while keeping your various components reusable across other types of applications. Finally, you'll learn how to keep your MauiProgram class maintainable by employing extension methods.

Paul D. Sheriff
http://www.pdsa.com

Paul has been working in the IT industry since 1985. In that time, he has successfully assisted hundreds of companies' architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has multiple courses in the www.pluralsight.com library (https://bit.ly/3gvXgvj) and on YouTube.com (https://www.youtube.com/@pauldsheriff) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

Introduction to Model-View-View-Model (MVVM)

You've been using an entity class to supply the controls with data, but an entity class shouldn't be used for this purpose. A **View Model** class is used to bind data to the controls on the UI. View models have properties used to set which buttons are enabled/visible, which menus are enabled/visible, and for posting informational and error messages. View model classes have methods to load data into entities or save data. View model classes may expose an entity object or have properties to expose only those properties of the entity object needed for the UI.

The whole point of using the MVVM design pattern is better reusability, maintainability, and testability of classes. Design your classes and assemblies in such a manner that you achieve the ability to reuse those classes across multiple projects, as shown in **Figure 1**. The "View Layer" such as WPF, Blazor, .NET MAUI, etc., should know how to use the properties and methods of the view model classes. However, the "View Model Layer" should not know about the view layer. The view model classes should know which repositories (Data Layer) and Model (Entity Layer) classes it can use to fill in its properties. However, the repositories and entities shouldn't know anything about the view model classes or the view layer. It's important to keep the classes in the view model, model, and data layer assemblies' technology-agnostic. In other words, those assemblies should be simple class libraries and have no references to any specific front-end technology to achieve maximum reusability.

Add More Properties to the Common Base Class

For the application being developed in this article, a few view model classes are needed. Typically, you create a view model class for each view/page you create in your application. Almost all pages in your application need to display information, error, and validation messages to the user. Let's add some properties to the CommonBase class to hold that data. Open the **BaseClasses\CommonBase.cs** file and add a couple of Using statements.

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
```

Add a string property named **InfoMessage** to hold informational messages, and another string property named **LastErrorMessage** to hold the last error message to display to the user. The **LastException** property keeps track of the last exception generated. The constant REPO_NOT_SET is used when you use a repository class to get data from a data store. This constant provides the error message to display if you forget to set the repository object into a class that needs it. You'll see examples of these later, but for now, add the code shown in **Listing 1** to the CommonBase.cs file.

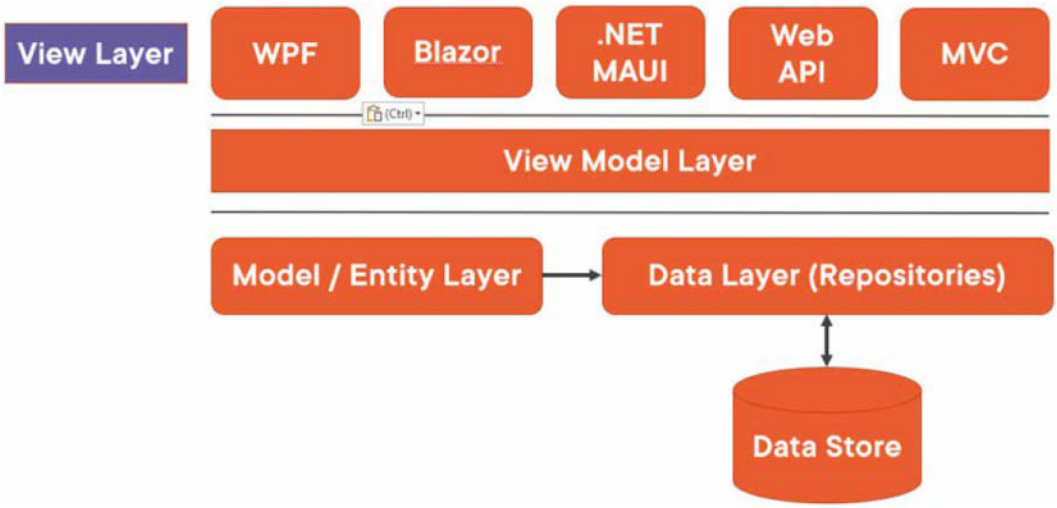


Figure 1: Use the MVVM design pattern for better reusability and maintainability.

Listing 1: Add message properties to the CommonBase class.

```
#region Private/Protected Variables
private string _InfoMessage = string.Empty;
private string _LastErrorMessage = string.Empty;
private Exception? _LastException = null;

protected const string REPO_NOT_SET =
  "The Repository Object is not Set.";
#endregion

#region Public Properties
[NotMapped]
[JsonIgnore]
public string InfoMessage {
  get { return _InfoMessage; }
  set {
    _InfoMessage = value;
    RaisePropertyChanged(nameof(InfoMessage));
  }
}

[NotMapped]
[JsonIgnore]
public string LastErrorMessage {
  get { return _LastErrorMessage; }
  set {
    _LastErrorMessage = value;
    RaisePropertyChanged(
      nameof(LastErrorMessage));
  }
}

[NotMapped]
[JsonIgnore]
public Exception? LastException {
  get { return _LastException; }
  set {
    _LastException = value;
    LastErrorMessage = _LastException == null
      ? string.Empty : _LastException.Message;
    RaisePropertyChanged(nameof(LastException));
  }
}
#endregion
```

Notice that each property raises the PropertyChanged event when a new value is set into the property. Two attributes are added to each property [NotMapped] and [JsonIgnore]. The [NotMapped] attribute is used because each entity class inherits from this class and if you use the Entity Framework, you don't want it thinking it needs to locate a column with any of these property names. The [JsonIgnore] attribute is optional, but you typically don't want to send these properties as JSON when serializing and deserializing an entity object through a Web API.

Create a View Model Base Class

Like you did with the entity classes, create a view model base class that each view model inherits from. Right mouse-click on the **Common.Library\BaseClasses** folder and add a new class named **ViewModelBase** and replace the entire contents of the new file with the code shown in **Listing 2**.

This class inherits from the CommonBase class, then adds an additional property and method. The **RowsAffected** property is set after each method that either retrieves or modifies data in a data store. This can be useful to display to the user how many items were found when searching, or how many items were modified. The **PublishException**() method is where you write code to perform logging of the exception.

Create View Model Layer Class Library

As you did with the User class, create a new Class Library project into which you place all your view model classes. Right mouse-click on the **Solution** and add a new Class Library project named **AdventureWorks.ViewModelLayer**. Delete the **Class1.cs** file as this file is not used. Right mouse-click on the **Dependencies** folder in this new **AdventureWorks.ViewModelLayer** project and add a project reference to the **Common.Library** project and to the **AdventureWorks.EntityLayer** project.

Add a User View Model Class

Right mouse-click on the **AdventureWorks.ViewModelLayer** project and add a new folder named **ViewModelClasses**. Right mouse-click on the **ViewModelClasses** folder and add a new class named **UserViewModel**. Replace the entire contents of this new file with the code shown in **Listing 3**.

Listing 2: Create a view model base class from which all your view models inherit.

```
namespace Common.Library;

public class ViewModelBase : CommonBase {
  #region Private Variables
  private int _RowsAffected;
  #endregion

  #region Public Properties
  public int RowsAffected
  {
    get { return _RowsAffected; }
    set
    {
      _RowsAffected = value;
      RaisePropertyChanged(nameof(RowsAffected));
    }
  }
  #endregion

  #region PublishException Method
  protected virtual void PublishException(Exception ex)
  {
    LastException = ex;

    System.Diagnostics.Debug.WriteLine(ex.ToString());
  }
  #endregion
}
```

The UserViewModel class (**Listing 3**) inherits from the ViewModelBase class you just created. It contains a property named **CurrentEntity** that's a User data type. The CurrentEntity property is used to bind each property in the User class to the appropriate controls on the UserDetailView page. There are three methods in this view model class; **GetAsync**(), **GetAsync**(**id**), and **SaveAsync**(). These methods do not currently interact with any data store, but later, you'll hook these up to different repository classes that get data from different data stores.

The GetAsync() method returns a list of users that can be bound to a collection-type view on the UserList page. For now, this method returns an empty list, but you'll add a

Listing 3: Create a user view model class to which to bind your controls.

```
using AdventureWorks.EntityLayer;
using Common.Library;
using System.Collections.ObjectModel;

namespace AdventureWorks.ViewModelLayer;

public class UserViewModel : ViewModelBase {
  #region Private Variables
  private User? _CurrentEntity = new();
  #endregion

  #region Public Properties
  public User? CurrentEntity  {
    get { return _CurrentEntity; }
    set {
      _CurrentEntity = value;
      RaisePropertyChanged(
        nameof(CurrentEntity));
    }
  }
  #endregion

  #region GetAsync Method
  public async
    Task<ObservableCollection<User>> GetAsync()
  {
    return await Task.FromResult(
      new ObservableCollection<User>());
  }
  #endregion

  #region GetAsync(id) Method
  public async Task<User?> GetAsync(int id) {
    try {
      // TODO: Get a User from a data store

      // MOCK Data
      CurrentEntity = await
        Task.FromResult(new User {
          UserId = id,
          LoginId = "SallyJones615",
          FirstName = "Sally",
          LastName = "Jones",
          Email = "Sallyj@jones.com",
          Phone = "615.987.3456",
          PhoneType = "Mobile",
          IsFullTime = true,
          IsEnrolledIn401k = true,
          IsEnrolledInFlexTime = false,
          IsEnrolledInHealthCare = true,
          IsEnrolledInHSA = false,
          IsEmployed = true,
          BirthDate =
            Convert.ToDateTime("08-13-1989"),
          StartTime = new TimeSpan(7, 30, 0)
        });

      RowsAffected = 1;
    }
    catch (Exception ex) {
      RowsAffected = 0;
      PublishException(ex);
    }

    return CurrentEntity;
  }
  #endregion

  #region SaveAsync Method
  public async virtual Task<User?> SaveAsync() {
    // TODO: Write code to save data
    System.Diagnostics.Debugger.Break();

    return await Task.FromResult(new User());
  }
  #endregion
}
```

list of user data soon. The GetAsync(id) method sets the CurrentEntity property to a single user object, then returns that object. The user object is hard coded currently, but later this data will come from a data store. The SaveAsync() method will eventually be called from the Save button on the user page, and for now, just returns a new user.

Use the User View Model on XAML

It's now time to change the user detail view to use the UserViewModel class instead of the User class. Right mouse-click on the **Dependencies** folder in this new **AdventureWorks.MAUI** project and add a project reference to the **AdventureWorks.ViewModelLayer** project. Open the **Views\UserDetailView.xaml** file and **change** the XML namespace "vm" to point to the ViewModelLayer assembly, as shown in the following code snippet.

```
xmlns:vm="clr-namespace:
  AdventureWorks.ViewModelLayer;
  assembly=AdventureWorks.ViewModelLayer"
```

Change the **x:DataType** attribute on the ContentPage to use the UserViewModel instead of the User class.

```
x:DataType="vm:UserViewModel"
```

Remove the **<vm:User x:Key="viewModel" ...>** from the <ContentPage.Resources> element as you're now going to use the hard-coded data returned from the GetAsync(id) method in the UserViewModel class. Remove the **BindingContext** attribute from the <Border> element as the BindingContent is going to be set in the code behind. Your <Border> element should now look like the following XAML.

```
<Border Style="{StaticResource Border.Page}">
```

Change All Bindings to Use the CurrentEntity Property

Because you've now wrapped up the User object within the CurrentEntity property on the UserViewModel class, change all the Binding markup extensions to use **CurrentEntity** before the property names, as shown in the following XAML. You should be able to do this with a search and replace in the Visual Studio editor.

```
{Binding CurrentEntity.PROPERTY_NAME}
```

Modify the Code Behind

Open the **Views\UserDetailView.xaml.cs** file and **change** the **using** statement that points to the EntityLayer to point to the ViewModelLayer instead, as shown in the following code snippet:

```
using AdventureWorks.ViewModelLayer;
```

Remove the line of code from the constructor that set the ViewModel to the value coming from the **Resources** collection. That was the view model created in the XAML

that's no longer there. Your constructor should now look like the following:

```
public UserDetailView() {
  InitializeComponent();
}
```

Change the **_ViewModel** variable to a **UserViewModel** data type as shown in the following code snippet. Create a **new** instance of the UserViewModel on the same line as the declaration.

```
private readonly UserViewModel _ViewModel = new();
```

Modify the OnAppearing() event procedure to look like the following code. Make sure you add the **async** keyword to the OnAppearing() event procedure. Set the BindingContext for the entire ContentPage to the **_ViewModel** variable. Call the GetAsync(id) method on the view model asynchronously to have the single instance of the User object placed into the CurrentEntity property.

```
protected async override void OnAppearing() {
  base.OnAppearing();

  // Set the BindingContext to the ViewModel
  BindingContext = _ViewModel;

  // Retrieve a User
  await _ViewModel.GetAsync(1);
}
```

Try It Out

Run the application and click on **Users > Navigate to Detail**. You should see that the hard-coded User data from the view model class is now bound to the controls on this user detail page.

Load Phone Picker Using MVVM

If you remember, the phone type list is a XAML resource located in the AppStyles.xaml file. Let's create a list of phone types in the user view model class so that list can be bound on the page. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable that's an **ObservableCollection** of strings. An ObservableCollection in .NET raises the PropertyChanged event whenever the collection is modified in any manner. This collection type should be used for all collections when working with .NET MAUI or WPF.

```
private ObservableCollection<string>
  _PhoneTypesList = [];
```

Let's expose this list of phone types so it can be bound on the user detail page. Add a public property named **PhoneTypesList** to the UserViewModel class as shown in the code snippet below.

```
public ObservableCollection<string>
  PhoneTypesList {
  get { return _PhoneTypesList; }
  set {
    _PhoneTypesList = value;
    RaisePropertyChanged(nameof(PhoneTypesList));
  }
}
```

In the UserViewModel class create a method named **GetPhoneTypes**() to load the list of phone types, as shown in the following code. All data is hard coded for now, but later in this article series, you'll get data from a data store.

```
#region GetPhoneTypes Method
public async
  Task<ObservableCollection<string>>
    GetPhoneTypes() {
  PhoneTypesList = await Task.FromResult(
    new ObservableCollection<string>(
      ["Home", "Mobile", "Work", "Other"])
    );

  return PhoneTypesList;
}
#endregion
```

Modify the User Detail View

Now that you have a method to load phone types, that method needs to be called before the user detail page is displayed. Open the **Views\UserDetailView.xaml.cs** file and add a call to the GetPhoneTypes() method in the **OnAppearing** event procedure, as shown in the following code.

```
protected async override void OnAppearing() {
  base.OnAppearing();

  // Set the Page BindingContext
  BindingContext = ViewModel;

  // Get the Phone Types
  await _ViewModel.GetPhoneTypes();

  // Retrieve a User
  await _ViewModel.GetAsync(1);
}
```

Because you're now getting the phone types from the view model class, you no longer need the XAML array. Open the **Resources\Styles\AppStyles.xaml** file and delete the **<x:Array x:Key="phoneTypes" ...>** element from the <ResourceDictionary> element. Open the **Views\UserDetailView.xaml** file and locate the <Picker> and change it to look like the following XAML.

```
<Picker VerticalTextAlignment="Center"
  SelectedItem="{Binding CurrentEntity.PhoneType}"
  ItemsSource="{Binding PhoneTypesList}" />
```

The **SelectedItem** property is bound to the **PhoneType** property on the user object so if the selection in the Picker control is changed, the **PhoneType** property in the user object is updated with the new value. The **ItemsSource** property is changed from the StaticResource markup extension to use the Binding markup extension that points to the observable collection of phone types in the UserViewModel class.

Try It Out

Run the application, click on **User > Navigate to Detail** and you should still see the same list of phone types, and the picker should be positioned on the value for the user read from the view model class.

Introduction to Dependency Injection

I want you to think about the <ShellContent> elements you created in the AppShell.xaml file. Each ShellContent object uses a **DataTemplate** markup extension to which you pass the name of a ContentPage you want to be displayed when clicked upon. You don't have to write any C# code to create the ContentPage; the .NET MAUI navigation engine takes care of all the details of page creation.

Think of some of the most common things you do when you're writing a typical business application. You probably use a tool like Log4Net or Serilog to record exceptions, information, and debug messages. You probably have an Entity Framework (EF) database context object for working with your database. You might have a set of repository classes that are responsible for working with each table in your database. You might also have a configuration manager class to retrieve application-wide settings from a JSON file, or maybe from a database.

The reason I'm pointing out these items is that I want you think about how many times in your application you might be instantiating your logging, your database context, your repository, or your configuration classes. If you have even a medium size application, the answer is that you're probably instantiating these classes in hundreds of locations throughout your code base. Now, think of the maintenance nightmare you'd encounter if you needed to replace the logging class with a different one, or that all your repository classes can't use a database context anymore, but need to make Web API calls. If you don't program with Dependency Injection, a change like this could cause hundreds of hours of re-work.

Dependency Injection (DI) helps you remove coupling between classes. In order to use DI, create an interface (or base class) for each of your logging, configuration, repository, database context, and other classes to implement or inherit. Now, instead of creating an instance of Serilog in each class that needs to perform logging, register Serilog with a DI service in the MauiProgram class. Most loggers today implement the ILogger interface. In each class that needs to use logging, you add the ILogger interface as an argument to the constructor of that class. You do the same process with repository, configuration, and even the EF DbContext class, as shown in **Figure 2**.

Register each interface and the corresponding class with the **Services** collection of the **MauiAppBuilder** object in the CreateMauiApp() method in the MauiProgram class, as shown in the following code snippet:

```
builder.Services.AddScoped
  <IRepository<User>, UserRepository>();
builder.Services.AddScoped
  <IRepository<PhoneType>, PhoneTypeRepository>();
```

In this code, you specify a service lifetime (Scoped, Singleton, Transient) and pass the interface followed by the class that implements that interface. Service lifetimes will be explained shortly. Each of these classes are placed into the DI container and are then ready to be used. A class is used when an instance of a class is created by .NET MAUI and that class has a constructor expecting a parameter of an interface type registered in the DI container.

In addition to classes that implement interfaces you also register view models and views if they need to receive any of the classes from DI to do their work. For example, if the

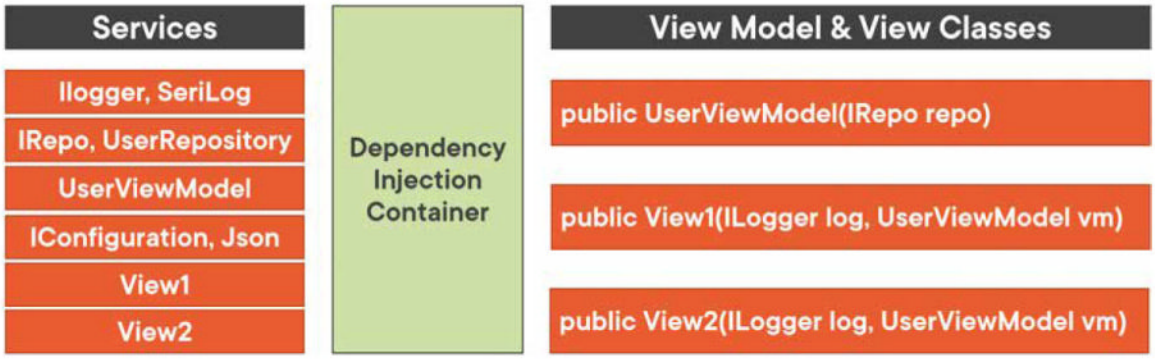


Figure 2: Services are placed into a DI container waiting to be referenced by an instance of a class.

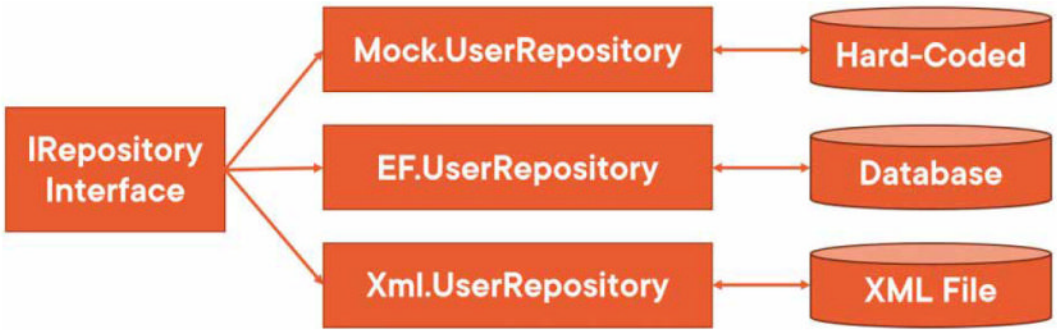


Figure 3: Use an interface to create different implementations of a service.

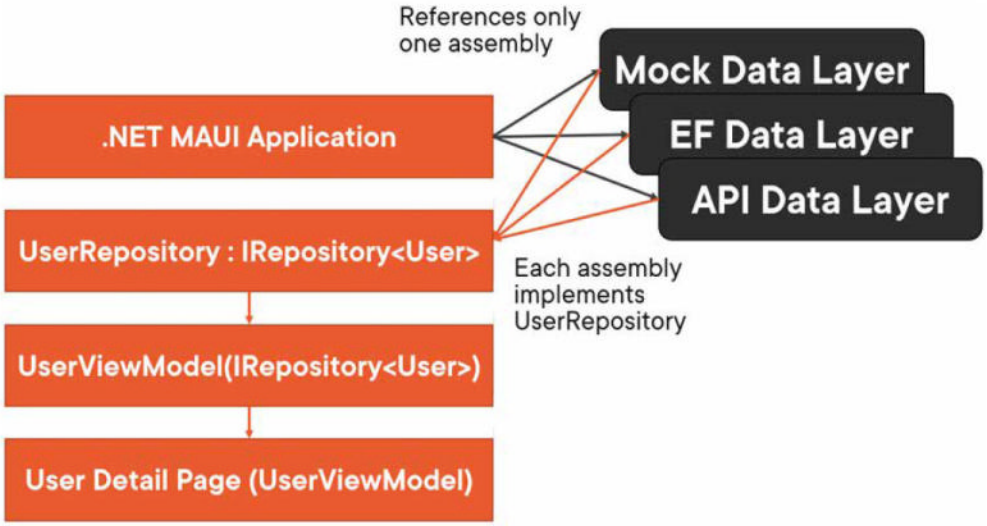


Figure 4: The .NET MAUI application selects one assembly with the implementation of an interface to use.

constructor of the UserDetailView has an IRepository<User> as an argument (shown in the following code) the navigation engine asks the DI container to look up the interface to see if it's registered. If it is, an instance of the UserRepository class is created by the DI engine and passed to the **repo** argument when the UserDetailView class is instantiated.

```
public UserDetailView(IRepository<User> repo) {
  InitializeComponent();

  // DO SOMETHING WITH THE REPOSITORY CLASS
}
```

Use Interfaces for Repository Classes

In a typical business application, you build a set of repository classes to talk to a data store to get and modify data. Each repository class needs certain standard methods such as Get(), Get(id), Insert(), Update(), Delete(), etc., that manipulate the data in a single table of your data store. Each repository class implements the interface by creating each method to perform the specified logic. If you have three tables in your application, such as Product, User, and PhoneTypes, you create three repository classes to implement each of the methods for performing CRUD logic.

For each repository class, you might start out using EF and a DbContext object to manipulate the data in each of these tables. However, when testing applications, you might want to use mock data instead. If you use .NET MAUI for a cross-platform application, each repository class should call Web APIs to perform the data manipulation. Each of these different types of classes implements the same interface, but each performs it differently.

Place each of these different repository implementations into different assemblies (**Figure 3**) and you reference just one of those assemblies at a time in your .NET MAUI program (**Figure 4**). The repository classes from the referenced assembly are the ones registered with the DI container in the MauiProgram class. As long as each view model and/or view only uses the repository interface, it doesn't matter which assembly you use, as they are completely interchangeable.

DI Service Lifetimes

The **Services** collection on the MauiAppBuilder object has three different methods you can use to register classes into the DI container; AddSingleton(), AddScoped(), and AddTransient(). When you add a class as a **Singleton,** the first time that class is requested by a constructor, the object is created and has a scope of the entire lifetime of the application. If another class requests that class, the exact same instance of the object first created is used. Because a .NET MAUI application is inherently single user, this type of lifetime is fine for views and view model classes.

When you add a class as **Scoped**, an instance of that class is created within the scope of the object that created it. When the object that created it is destroyed, this object is also destroyed. This type of lifetime is good for repository objects because if you have a DbContext object, you don't want that object to stay around any longer than necessary. This type of lifetime is appropriate for views and view models.

A class added as **Transient** is like a scoped object, but there's no pre-defined lifetime. It may be released anytime, and if requested again, a new instance is created. This type of lifetime is good for classes that log data or read configuration items.

Add an IRepository Interface

Right mouse-click on the **Common.Library** project and add a new folder named **Interfaces**. Right mouse-click on the new **Interfaces** folder and add a new file named **IRepository**. Replace the entire contents of the new file with the following code:

```
using System.Collections.ObjectModel;

namespace Common.Library;

public interface IRepository<TEntity> {
  Task<ObservableCollection<TEntity>> GetAsync();
  Task<TEntity?> GetAsync(int id);
}
```

Listing 4: Use a Repository object to retrieve data from a data store.

```
public async Task<User?> GetAsync(int id) {
  try {
    // Get a User from a data store
    if (_Repository != null) {
      CurrentEntity = await _Repository.GetAsync(id);
      if (CurrentEntity == null) {
        InfoMessage = $"User id={id} was not found.";
      }
      else {
       InfoMessage = "Found the User";
      }
    }
    else {
      LastErrorMessage = REPO_NOT_SET;
      InfoMessage = "Found a MOCK User";

      // MOCK Data
      CurrentEntity =
        await Task.FromResult(new User {
        UserId = id,
        LoginId = "SallyJones",
        FirstName = "Sally",
        LastName = "Jones",
        Email = "Sallyj@jones.com",
        Phone = "615.987.3456",
        PhoneType = "Mobile",
        IsEnrolledIn401k = true,
        IsEnrolledInFlexTime = false,
        IsEnrolledInHealthCare = true,
        IsEnrolledInHSA = false,
        IsEmployed = true,
        BirthDate
          = Convert.ToDateTime("08-13-1989"),
        StartTime = new TimeSpan(7, 30, 0)
      });
    }

    RowsAffected = 1;
  }
  catch (Exception ex) {
    RowsAffected = 0;
    PublishException(ex);
  }

  return CurrentEntity;
}
```

The generic IRepository<TEntity> interface requires you to identify the Entity class name when declaring a variable. For example, IRepository<User> or IRepository<Product> identifies the type of object(s) returned from the GetAsync() or GetAsync(id) methods. All the Repository classes you build in this article series are going to be asynchronous, so all method names are going to reflect this. Other methods to support data modification will be added later to this interface.

Use the IRepository Interface in the User View Model Class

Let's now use the IRepository interface in the user view model class. The Common.Library project is already referenced from the view AdventureWorks.ViewModelLayer project, so the IRepository interface is available to use. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable that is of the type IRepository<User>, as shown in the following line of code:

```
private readonly IRepository<User>? _Repository;
```

Add two constructors, one empty, and the other injects the IRepository<User> interface, as shown in the following code. Assign the **repo** variable to the private read-only **_Repository** variable. You haven't built this repository class yet, but you don't need it to create the appropriate code in the user view model class to make calls to the methods defined in the interface.

```
#region Constructors
public UserViewModel() {
}

public UserViewModel(IRepository<User> repo)
  : base() {
  _Repository = repo;
}
#endregion
```

Use the IRepository Interface in View Model Classes

Modify the **GetAsync(id)** method (**Listing 4**) to retrieve a single user by calling the GetAsync(id) method of the **_Repository** variable. Of course, you're going to ensure that this object is of the type UserRepository a little later. In the code shown in **Listing 4,** check to see if the **_Repository** variable is not null. If it's not null, make a call to the GetAsync(id) on the repository class. The value returned is placed into the **CurrentEntity** property. Once the **CurrentEntity** property is set and the PropertyChanged event fires, the UI re-reads the data and updates all bindings on the page.

Also notice that the code sets the **LastErrorMessage** property if the **_Repository** variable has not been set. The **InfoMessage** property is set to one of two messages depending on whether the user ID was located or not. If mock data is returned because the **_Repository** variable was null, then the **InfoMessage** property is set to tell the user that mock data was used. Finally, the **RowsAffected** property is set to a value of one (1).

Try It Out

Run the application and click on **Users < Navigate to Detail** to see the hard-coded user coming from this new method. The messages do not show up anywhere, but those will be added later in this article series.

Create a Data Layer Class Library

As stated before, you may want to have different data stores from which you get data. I'd highly recommend that you create different class library projects for each of these different data stores. In this article, you're going to use mock data. Right mouse-click on the **Solution** and add a new **Class Library** project named **AdventureWorks.DataLayer.Mock**. Delete the **Class1.cs** file, as this file isn't needed. Right mouse-click on the **Dependencies** folder in the **AdventureWorks.DataLayer.Mock** project and add two project references to the **AdventureWorks.EntityLayer** and **Common.Library** class libraries.

Add a User Repository Class

Right mouse-click on the new **AdventureWorks.DataLayer.Mock** project and add a folder named **RepositoryClasses**. Right mouse-click on the **RepositoryClasses** folder and

Listing 5: Create a repository class to return a set of mock user data.

```
using AdventureWorks.EntityLayer;
using Common.Library;
using System.Collections.ObjectModel;
using System.Data;

namespace AdventureWorks.DataLayer;

/// <summary>
/// Create a set of User mock data
/// </summary>
public partial class UserRepository
  : IRepository<User> {
  #region GetAsync Method
  public async
    Task<ObservableCollection<User>>
      GetAsync() {
    return await Task.FromResult(
      new ObservableCollection<User>()
    {
      new() {
        UserId = 1,
        LoginId = @"MichaelThomason",
        FirstName = @"Michael",
        LastName = @"Thomason",
        Email = @"MichaelThomason@advworks.com",
        Phone = @"615.555.3333",
        PhoneType = @"Mobile",
        IsFullTime = true,
        IsEnrolledIn401k = true,
        IsEnrolledInHealthCare = true,
        IsEnrolledInHSA = false,
        IsEnrolledInFlexTime = false,
        IsEmployed = true,
        BirthDate = new DateTime(1985, 3, 15),
        StartTime = null,
      },
      new() {
        UserId = 2,
        LoginId = @"SheilaCleverly",
        FirstName = @"Sheila",
        LastName = @"Cleverly",
        Email = @"SheilaCleverly@advworks.com",
        Phone = @"615.123.3456",
        PhoneType = @"Other",
        IsFullTime = false,
        IsEnrolledIn401k = false,
        IsEnrolledInHealthCare = true,
        IsEnrolledInHSA = false,
        IsEnrolledInFlexTime = false,
        IsEmployed = true,
        BirthDate = new DateTime(1981, 6, 9),
        StartTime = new TimeSpan(7, 30, 0),
      },
      new() {
        UserId = 3,
        LoginId = @"CatherineAbel",
        FirstName = @"Catherine",
        LastName = @"Abel",
        Email = @"CatherineAbel@advworks.com",
        Phone = @"615.998.3332",
        PhoneType = @"Mobile",
        IsFullTime = true,
        IsEnrolledIn401k = true,
        IsEnrolledInHealthCare = true,
        IsEnrolledInHSA = true,
        IsEnrolledInFlexTime = true,
        IsEmployed = true,
        BirthDate = new DateTime(1979, 4, 5),
        StartTime = null,
      }

      // ADD MORE MOCK DATA HERE

    });
  }
  #endregion

  #region GetAsync(id) Method
  public async Task<User?> GetAsync(int id) {
    ObservableCollection<User> list =
      await GetAsync();
    User? entity = list.Where(
      row => row.UserId == id).FirstOrDefault();

    return entity;
  }
  #endregion
}
```

add a new class named **UserRepository**. Replace the entire contents of this new file with the code shown in **Listing 5**. The UserRepository class implements the IRepository interface and thus contains the implementation of the two methods GetAsync() and GetAsync(id). The GetAsync() method creates a list of three users and returns them as an ObservableCollection of User objects. The GetAsync(id) method retrieves the list of users from the GetAsync() method, then applies the LINQ Where() method to the list to locate the ID passed in as the parameter.

Add Dependency in MAUI Project

As mentioned previously, you need one concrete implementation of a data layer in your .NET MAUI application. Right mouse-click on the **Dependencies** folder in the **AdventureWorks.MAUI** project and a project reference to **AdventureWorks.DataLayer.Mock** project. Do **NOT** add a reference to the DataLayer.Mock project to the view model layer. All the view model class needs to know is the interface that the repository classes are using. The concrete implementation of the repository classes is created by .NET MAUI's DI engine and those implementations are injected into the view models.

Inject View Models into Views

You now have the IRepository<User> being passed to the constructor of the UserViewModel class. You need to pass

Listing 6: Create a PhoneType entity class.

```
using Common.Library;

namespace AdventureWorks.EntityLayer;

public class PhoneType : EntityBase {
  #region Private Variables
  private int _PhoneTypeId;
  private string _TypeDescription
    = string.Empty;
  #endregion

  #region Public Properties
  public int PhoneTypeId {
    get { return _PhoneTypeId; }
    set {
      _PhoneTypeId = value;
      RaisePropertyChanged(nameof(PhoneTypeId));
    }
  }

  public string TypeDescription {
    get { return _TypeDescription; }
    set {
      _TypeDescription = value;
      RaisePropertyChanged(
       nameof(TypeDescription));
    }
  }
  #endregion
}
```

Listing 7: Modify the GetPhoneTypes method to use the repository to retrieve the list of phone types.

```
#region GetPhoneTypes Method
public async Task<ObservableCollection<string>>
  GetPhoneTypes() {
  if (_PhoneTypeRepository != null) {
    var list =
      await _PhoneTypeRepository.GetAsync();

    PhoneTypesList = new
      ObservableCollection<string>
      (list.Select(row => row.TypeDescription));
  }

  return PhoneTypesList;
}
#endregion
```

an instance of the UserViewModel class to the UserDetailView page so the page can bind to the properties of the view model class, and make calls to any methods. Open the **Views\UserDetailView.xaml.cs** file and modify the constructor to accept the view model class and to assign that variable to the **_ViewModel** variable.

```
public partial class UserDetailView
  : ContentPage {
  public UserDetailView(UserViewModel viewModel) {
    InitializeComponent();

    _ViewModel = viewModel;
  }

  // REST OF THE CODE HERE
}
```

Register Classes with DI Container

The responsibility for creating an instance of the UserDetailView is through the .NET MAUI navigation system. So, how do the UserViewModel and the UserRepository classes get passed to the UserDetailView and the UserViewModel respectively? Register these classes in the MauiProgram class using the MauiAppBuilder object. Open the **MauiProgram.cs** file and add some new using statements.

```
using AdventureWorks.DataLayer;
using AdventureWorks.EntityLayer;
using AdventureWorks.MAUI.Views;
using AdventureWorks.ViewModelLayer;
using Common.Library;
```

Just below the **builder.UseMauiApp<App>()**, add the following code to inject the repository, the view model, and the view into the DI container.

```
// Add classes for use in Dependency Injection
builder.Services.AddScoped<IRepository<User>,
  UserRepository>();
builder.Services.AddScoped<UserViewModel>();
builder.Services.AddScoped<UserDetailView>();
```

Try It Out

Run the application and click on the **User > Navigate to Detail** menu to see the user associated with the **UserId** property equal to one (1) in the user repository appear. Stop the application and try using one of the other user ID values such as 2 or 3.

Load Phone Types Using Repository Class

Earlier in this article, you loaded the phone types by creating a collection in the UserViewModel class. Instead of hard coding these types in the UserViewModel class, create a PhoneType entity class, and a PhoneTypeRepository class to retrieve the phone type data from a data store. Creating entity and repository classes for phone types allows you to reuse them on other pages.

Add a PhoneType Entity Class

Right mouse-click on the **EntityClasses** folder in the **AdventureWorks.EntityLayer** project and add a new

Listing 8: Add a repository class to work with PhoneType objects.

```
using AdventureWorks.EntityLayer;
using Common.Library;
using System.Collections.ObjectModel;

namespace AdventureWorks.DataLayer;

/// <summary>
/// Create a set of PhoneType mock data
/// </summary>
public partial class PhoneTypeRepository
   : IRepository<PhoneType> {
  #region GetAsync Method
  public async
    Task<ObservableCollection<PhoneType>>
      GetAsync() {
    return await Task.FromResult(
      new ObservableCollection<PhoneType>
    {
      new() {
         PhoneTypeId = 1,
         TypeDescription = "Home",
      },
      new() {
         PhoneTypeId = 2,
         TypeDescription = "Mobile",
      },
      new() {
         PhoneTypeId = 3,
         TypeDescription = "Work",
      },
      new() {
         PhoneTypeId = 4,
         TypeDescription = "Other",
      }
    });
  }
  #endregion

  #region GetAsync(id) Method
  public async Task<PhoneType?>
    GetAsync(int id) {
    ObservableCollection<PhoneType> list
      = await GetAsync();
    PhoneType? entity = list.Where(
       row => row.PhoneTypeId == id)
       .FirstOrDefault();

    return entity;
  }
  #endregion
}
```

Listing 9: Create a Commanding View Model class in your .NET MAUI application.

```
using AdventureWorks.EntityLayer;
using Common.Library;
using System.Windows.Input;

namespace AdventureWorks.MAUI.MauiViewModelClasses;

public class UserViewModel :
  AdventureWorks.ViewModelLayer.UserViewModel {
  #region Constructors
  public UserViewModel() : base() {
  }

  public UserViewModel(IRepository<User> repo)
    : base(repo) {
  }

  public UserViewModel(IRepository<User> repo,
   IRepository<PhoneType> phoneRepo) :
     base(repo, phoneRepo) {
  }
  #endregion

  #region Commands
  public ICommand? SaveCommand
    { get; private set; }
  #endregion

  #region Init Method
  public override void Init() {
    base.Init();

    // Create commands for this view
    SaveCommand = new Command(
      async () => await SaveAsync());
  }
  #endregion
}
```

class named **PhoneType**. Replace the entire contents of this new file with the code shown in **Listing 6**. This class contains a phone type identifier and the description of the phone type. The same design pattern is followed in this class as in the User class. You use private variables for each property, then expose that private variable through getter and setter. Don't forget to add the call to the RaisePropertyChanged() event in the setter. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private variable of the type **PhoneTypeRepository,** as shown in the following line of code.

```
private readonly IRepository<PhoneType>?
  _PhoneTypeRepository;
```

Add a third constructor to accept an **IRepository<PhoneType>** object from the DI engine, as shown in the following code.

```
public UserViewModel(IRepository<User> repo,
  IRepository<PhoneType> phoneRepo) : base() {
  _Repository = repo;
  _PhoneTypeRepository = phoneRepo;
}
```

Locate the GetPhoneTypes() method and modify the code to look like **Listing 7**. The **_PhoneTypeRepository** private variable injected into the constructor is used to retrieve the list of phone types from the data store used by the PhoneTypeRepository class.

Add a Phone Type Repository Class

Right mouse-click on the **RepositoryClasses** folder in the **AdventureWorks.DataLayer.Mock** project and add a new class named **PhoneTypeRepository**. Replace the entire contents of this new file with the code shown in **Listing 8**. Once again, a familiar design pattern is applied to the PhoneTypeRepository class. Implement the IRepository interface and create the two methods GetAsync() and GetAsync(id).

Add Phone Types to DI

Open the **MauiProgram.cs** file in the **AdventureWorks.MAUI** project and add a new service to the DI container to use the **PhoneTypeRepository** class.

Try It Out

Run the application, click on **User > Navigate to Detail** and you should still see the same list of phone types, and the phone type picker should be positioned on the value for the user read from the repository.

Eliminate Event Handling with Commanding

The MVVM design pattern eliminates a lot of code from the UI layer by removing code behind pages and moving it down to a view model. On the user detail page, you have two buttons: Save and Cancel. You created an event procedure to handle the click event on the Save button, as shown in the following code.

```
private void SaveButton_Clicked(object sender,
  EventArgs e) {
  // TODO: Respond to the event here
  System.Diagnostics.Debugger.Break();
}
```

The SaveButton_Clicked event procedure is mapped to the Save button by setting the Clicked attribute as shown below.

```
<Button Text="Save"
        Clicked="SaveButton_Clicked" />
```

As you can imagine, as you add more buttons to a page, your code behind can grow significantly. Eliminating code behind helps make your code more portable if you wish to recreate your UI in a different technology. The more code you have in your view model, entity, and repository classes, the more your code is reusable.

The SaveButton_Clicked event procedure is eliminated by using a technique called **Commanding**. Commanding is where you create a property that is of the type **ICommand**. You then create an instance of this ICommand by instantiating a **Command** class. Once you've created this Command property, bind it to the Save button with code that looks like the following:

```
<Button Text="Save"
        Command="{Binding SaveCommand}" />
```

Listing 10: Add a class for extension methods of the IServicesCollection object.

```
using AdventureWorks.DataLayer;
using AdventureWorks.EntityLayer;
using AdventureWorks.MAUI.Views;
using AdventureWorks.ViewModelLayer;
using Common.Library;

namespace AdventureWorks.MAUI.ExtensionClasses;

public static class ServiceExtensions {
  public static void AddServicesToDIContainer(
    this IServiceCollection services) {
    // Add Repository Classes
    AddRepositoryClasses();
    // Add View Model Classes
    AddViewModelClasses(services);
    // Add View Classes
    AddViewClasses(services);
  }

  private static void AddRepositoryClasses(
    IServiceCollection services) {
    // Add Repository Classes
    services.AddScoped<IRepository<User>,
      UserRepository>();
    services.AddScoped<IRepository<PhoneType>,
      PhoneTypeRepository>();
  }

  private static void AddViewModelClasses(
    IServiceCollection services) {
    // Add View Model Classes
    services.AddScoped<UserViewModel>();
  }

  private static void AddViewClasses(
    IServiceCollection services) {
    // Add View Classes
    services.AddScoped<UserDetailView>();
  }
}
```

Getting the Sample Code

You can download the sample code for this article by visiting www.CODEMag.com under the issue and article, or by visiting www.pdsa.com/downloads. Select "Articles" from the Category drop-down. Then select "Exploring .NET MAUI: MVVM, DI, and Commanding" from the Item drop-down.

The ICommand interface is defined in the System.Windows.Input namespace, which is located in the System.ObjectModel assembly. The System.ObjectModel assembly is a part of the Microsoft.NETCore.App framework, so the ICommand can be used in your view model layer assembly if you wish. However, the concrete implementation of the Command class is located in the Microsoft.Maui.Controls project. You don't want to add this assembly into your view model layer as this couples the view model assembly to .NET MAUI and that can cause problems if you want to reuse the view model assembly in an MVC, a WPF, or a Blazor application.

To take advantage of commanding, create a view model class in the .NET MAUI application that inherits from the view model class in the view model layer project. The view model class created in the .NET MAUI application can implement the commanding, yet all the other reusable functionality is kept in the view model layer project.

Implement Commanding in Your Project

Right mouse-click on the **AdventureWorks.MAUI** project and add a new folder named **MauiViewModelClasses**. Right mouse-click on the **MauiViewModelClasses** folder and add a new class named **UserViewModel**. Replace the entire contents of this new file with the code shown in **Listing 9**. In the code shown in **Listing 9,** create three constructors just like you did in the UserViewModel in the ViewModelLayer project. Each of these constructors call the base constructor passing in the arguments. The **SaveCommand** property is defined as an ICommand interface. This property is initialized in the Init() method. The Init() method is called from the CommonBase class constructor so it doesn't need to be called from any of the view model constructors. The first parameter to the Command constructor is any anonymous method that says to call the SaveAsync() method located in the UserViewModel class in the ViewModelLayer project.

Change Your XAML File to Use the New View Model Class

You need to use the new view model class in your XAML files, instead of the view model class from the view model layer project. Open the **Views\UserDetailView.xaml** file and change the "**xmlns:vm**" namespace to reference the new namespace in the .NET MAUI application, as shown below.

```
xmlns:vm="clr-namespace:
  AdventureWorks.MAUI.MauiViewModelClasses"
```

Modify the **Save** button and remove the **Clicked** attribute, as shown in the code snippet below. Add a **Command** attribute and set the Binding markup extension to the SaveCommand property you created as show below. When the Save button is clicked upon, the SaveCommand is fired and the SaveAsync() method is invoked.

```
<Button Text="Save"
  ImageSource="save.png"
  ToolTipProperties.Text="Save Data"
  ContentLayout="Left"
  Command="{Binding SaveCommand}" />
```

Change the View Model Reference

Open the **MauiProgram.cs** file and remove the modify the DI injection to use the new MauiViewModelClasses.UserViewModel class instead of the view model coming from the view model layer project.

```
builder.Services.AddScoped<
  MauiViewModelClasses.UserViewModel>();
```

Modify the UserDetailView Code

Open the **Views\UserDetailView.xaml.cs** file and change the **using AdventureWorks.ViewModelLayer;** to use the new namespace, as shown below.

```
using AdventureWorks.MAUI.MauiViewModelClasses;
```

Remove the **SaveButton_Clicked()** event procedure, as this code is no longer needed.

Try It Out

Run the application, click on the **Users > Navigate to Detail** menu, click on the **Save** button, and you should end up on the Break() method in the SaveAsync() method in the UserViewModel class in the ViewModelLayer project. There are more options in the Command class that you can take advantage of, such as the ability to disable a button through a **CanExecute** property. I don't tend to use this property as I control disabling buttons in my view model by binding a Boolean view model property

to the **IsEnabled** property on the buttons. This makes the view model code easier to reuse in technologies that don't support commanding or don't implement commanding in the same manner.

Advantages of Commanding

Many times, the code in an event handler is just a single line of code calling a method in a view model. By using commanding, you directly call methods in the view model without the event handler code. Commanding thereby eliminates event handling in the user interface of applications making your applications more reusable, easier to test, and more maintainable.

Disadvantages of Commanding

Although there are great advantages to using commanding, there are some downsides to it as well. First and foremost, commanding isn't available natively in all UI technologies such as Windows Forms, Web Forms, or WPF. Many of the "Community Toolkits" for these technologies do provide some form of commanding. However, not all these toolkits implement commanding in the same way. You can solve a lot of these pitfalls by using the design pattern I presented in this article. Take advantage of inheritance and just write a little bit of code in the UI layer and leave your view model classes without any ties to a specific UI technology or toolkit.

Clean Up the MauiProgram Class

Open the **MauiProgram.cs** file in the AdventureWorks.MAUI project and you'll see that you have quite a few lines just adding repository, view model, and view classes into the DI container. As you can imagine, this list of services can get quite long and thus make your MauiProgram class unmanageable. The **builder.Services** object is an IServiceCollection data type, so let's create a set of extension methods for this type in a separate class.

Right mouse-click on the AdventureWorks.MAUI project and add a new folder named **ExtensionClasses**. Right mouse-click on the **ExtensionClasses** folder and add a new class named **ServiceExtensions**. Replace the entire contents of this new file with the code shown in **Listing 10**.

There's one public method in the ServiceExtensions class, AddServicesToDIContainer(), that's called from the MauiProgram.CreateMauiApp() method. This method calls three other private methods to add repository, view model, and view classes. If you have other types of classes to add to the DI container, add additional methods as appropriate. The ServiceExtension class accomplishes the same functionality as the lines of code currently in the MauiProgram.cs file but organizes the adding of services into different methods. Once you have the ServiceExtensions class created, eliminate the lines of code in the MauiProgram class that added classes to the Services collection with the code shown below.

```
// Add Classes to Dependency Container
builder.Services.AddServicesToDIContainer();
```

Go to the top of the **MauiProgram.cs** file and eliminate any unnecessary **using** statements.

Try It Out

Run the application to ensure that everything still works as it did before all the changes you just made.

Clean Up Warnings in Error List Window

While we're on the subject of cleaning up, let's also ensure that you're using the **x:DataType** attribute on all pages and partial pages. If you remember, the x:DataType attribute, when applied on a page, uses compiled bindings on that page. Compiled bindings improve the speed of data binding at runtime by resolving binding expressions at compile-time. Open the **ViewsPartial\HeaderView.xaml** file and add an XML namespace to the ViewsPartial namespace, as shown below.

```
xmlns:partial="clr-namespace:
  AdventureWorks.MAUI.ViewsPartial"
```

Add the x:DataType attribute, as shown in the following code snippet:

```
x:DataType="partial:HeaderView"
```

By adding these two lines, you eliminate some warnings in the Error List window. Not all can be eliminated yet, as you haven't created a view model for the product data. These warnings will eventually be eliminated as you continue on with this article series.

Summary

Using the Model-View-View-Model and the Dependency Injection design patterns helps you create reusable, maintainable, and testable code. Always be thinking how to keep common code in separate assemblies for maximum reusability. Commanding is a great way to reduce code behind, but ensures that you keep the code in the UI layer and not in your generic class libraries. Keep your MauiProgram.cs file organized by creating extension methods in a separate class. In the next article, you'll learn to display lists of data, select an item from the list, and navigate to the detail page for that item.

Paul D. Sheriff

SPONSORED SIDEBAR

Ready to Modernize a Legacy App?

Need advice on migrating yesterday's **legacy applications** to today's modern platforms? Take advantage of **CODE Consulting's** years of experience and contact us today to schedule a **FREE** consulting call to discuss your options.

No strings. No commitment.

For more information: www.codemag.com/consulting or email us at info@codemag.com.

Building Microservices Architecture Using CQRS and ASP.NET Core

In today's fast-paced business landscape, we often build scalable, secure, high-performing, and maintainable applications. A plethora of design patterns and architectural approaches can help in this regard. Command Query Reponsibility Segregation, or CQRS, is a proven architectural pattern that can help build scalable applications in complex scenarios. It does this by splitting responsibilities among read and write models. This article discusses the CQRS pattern, why it's important, and shows how you can implement the CQRS pattern in Microservices-based applications.

Joydip Kanjilal
joydipkanjilal@yahoo.com

Joydip Kanjilal is an MVP (2007-2012), software architect, author, and speaker with more than 20 years of experience. He has more than 16 years of experience in Microsoft .NET and its related technologies. Joydip has authored eight books, more than 500 articles, and has reviewed more than a dozen books.

If you're to work with the code examples discussed in this article, you need the following installed in your system:

- Visual Studio 2022
- .NET 9.0
- ASP.NET 9.0 Runtime

If you don't already have Visual Studio 2022 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

Understanding the Problem

Consider an enterprise application built in ASP.NET Core that needs to handle big data or massive amounts of data. For example, the application might be handling millions of complex transactions such as retrieving product details, updating stock, processing orders, etc. Over time, as the application attempts to scale to handle more concurrent requests, things can get complicated because you might be using the same models for data read and write operations. As a result, you might often observe inconsistences in your application's data.

The CQRS pattern can help you isolate these operations into commands (i.e., create, update, and delete data) and queries (i.e. retrieve data from the database). This helps you optimize and scale the command and query components of your application independently, enabling your application to be high performant, scalable, and reliable. **Figure 1** shows typical application-enabling queries and updates from and to an underlying database.

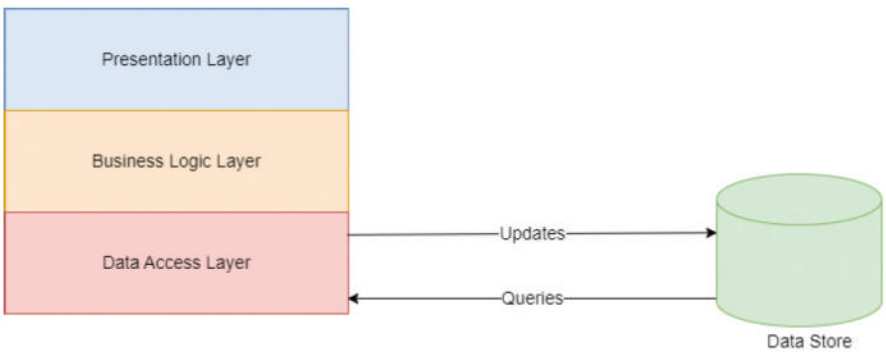


Figure 1: A typical application with queries and updates from and to a database

Why do you need a mediator? The mediator is the component in your application that's responsible for routing each request to the appropriate component. The result? Your application's code will be more lean, clean, decoupled, and manageable.

An Introduction to the CQRS Pattern

In most applications, the same model is used both for read and write/update operations. When you're using simple CRUD operations (create, read, update, and delete), you're good to use the same model to query data as well as save/update data. Over time, as the application grows and you have more and more data in the database, things become complicated. You might often observe anomalies in read and write operations because you have certain properties that must be persisted or updated in the database but you don't want them to be returned in queries.

For example, you might need the ProductId of the Product model to be stored in the database but not returned when the same model is queried. This might lead to data loss and inconsistencies in data. Enter the CQRS pattern. CQRS, an acronym for Command and Query Responsibility Segregation, is an architectural pattern in which the data read and data write operations are isolated from one another, as shown in **Figure 2**.

You can even isolate the read data and write data by using separate databases for each. In this case, you'll have two databases, one for reading or querying data optimized for queries and one write database that's optimized for create, update, and delete operations. Hence, your read database can be a document database and the write database can be a relational database. By isolating the read and write data stores, you can achieve enhanced scalability to handle increased loads.

For instance, you may want to optimize your read database to withstand significantly greater loads than your write database. There are more read operations in an application than writes and updates. **Figure 3** shows an implementation of the CQRS pattern.

Here's how the entire process works:

1. The client communicates with an application by sending commands using an API as an interface.
2. The application receives the command and processes it.
3. The application writes the data associated with the command into the write (or command) repository.

4. Once the command is saved to the write database, events are fired in the read (query) repository to update the data.
5. In the read (or query) database, the data persists after processing.
6. By communicating with the APIs used for receiving data, the client sends queries to the query side of the program.
7. The application processes the read request to retrieve the appropriate data from the read database.

> In CQRS, a command should always be task-based and not data-centric. For example, "Reserve a train ticket" is an example of a command. However, "Change train reservation status to Reserved" is not a command.

Benefits

Here are the key benefits of the CQRS pattern:

- **Separation of concerns:** The CQRS pattern separates the query (i.e., data retrieval without modifying the state) and command (i.e., create, update, and delete operations) components of the application, enabling you to optimize both independently. Although you can optimize your queries to be more efficient and fast, you can implement strict validation, transaction, and security logic in the command components.
- **Optimization:** Because the read and write operations are isolated into two different models, the CQRS pattern can help in optimizing performance of data access operations in your application. Although it can help you optimize query performance by improving the speed of data retrieval in read operations, the CQRS pattern can help you preserve transactional integrity and domain logic, and optimize write or update operations in the database.
- **Scalability:** Because the CQRS pattern splits the data access components into read and write components, it enables you to scale each of these components independently of one another. Typically, write operations in an application are fewer compared to read operations. Hence, if the application experiences heavy read traffic, the read models can be scaled horizontally.
- **Security:** The CQRS pattern helps you to implement different security strategies for read and write operations in your application. For example, you might want to secure certain operations that write or update sensitive data in the database. On the other hand, you may want most of the read operations in the application to allow data to the made available.
- **Maintainability:** Because it isolates the read and write operations in an application, the CQRS pattern facilitates maintainability due to separation of concerns. For example, you can change the query components of your application without affecting the command side of your application that's responsible for updating or persisting data.

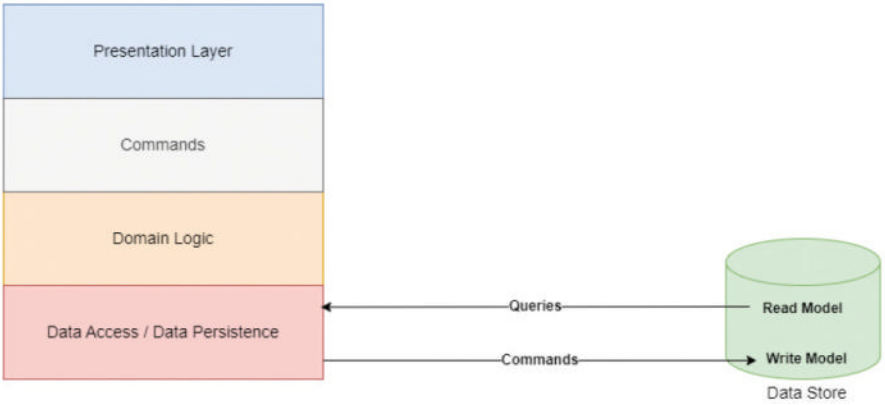


Figure 2: CQRS at work with the data read and write operations isolated

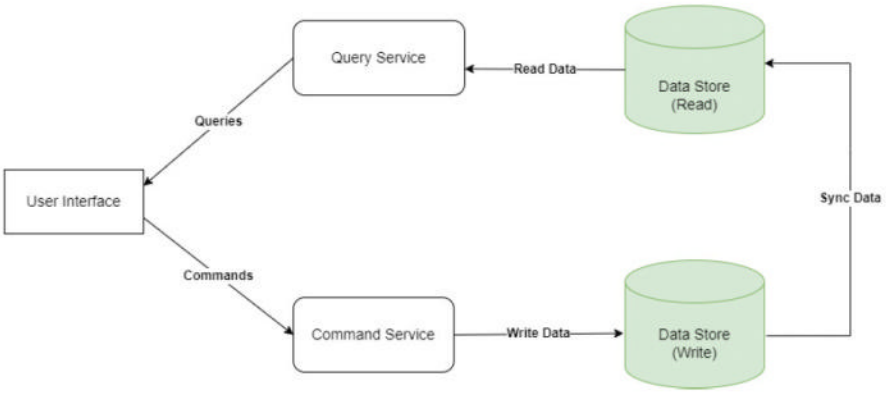


Figure 3: An implementation of the CQRS pattern with separate databases for read and write operations

Key Components of the CQRS Design Pattern

The CQRS pattern encompasses several key components:

- **Commands:** These are components that help you change the application's state. For example, you can take advantage of commands to create new data, update data, or delete data.
- **Queries:** These components don't change the application's state; instead, they help you in data retrieval from the data store.
- **Command handlers:** These are components that accept some incoming commands, perform the actions per these commands and consequently alter the application's state.
- **Query handlers:** These are components that can help you build queries, execute those queries to retrieve data out of a data store, and subsequently return this data to the invoker.

Challenges of the CQRS Design Pattern

There are several challenges of the CQRS pattern:

- **Increased complexity:** CQRS introduces additional complexity in your application because of the need for different paths for reading and writing/updating data. Additionally, applications that take advantage

of CQRS pattern should have different components for reading and writing operations, which can eventually become an overhead.

- **Consistency:** It's difficult to ensure that your application's data is consistent—updates to the data in your data store must be reflected in the query results. Hence, you need proper data synchronization mechanisms to ensure that the data pertaining to read and write operations is in sync. Ensure that there's no data duplication between command and query components. It's quite challenging to ensure data integrity across multiple data stores, particularly when there's a system or network failure.
- **Operational overhead:** The CQRS pattern can introduce operational overhead because of the need to deploy services for both query and write operations in separate servers. Deploying, scaling, monitoring, and debugging applications that take advantage of the CQRS pattern can also be challenging.
- **Testing and debugging:** It's quite challenging to test and debug applications that leverage the CQRS pattern. You might need to adopt specific testing strategies because of the asynchronous nature of the CQRS pattern. Additionally, because the commands and events are processed in an isolated manner and asynchronously, detecting issues in distributed applications can be challenging.

Use Cases

Typically, the CQRS pattern is used in large and complex projects where performance and scalability are important. Here are the key use cases of the CQRS pattern:

- E-commerce applications
- Healthcare applications
- Financial applications
- IoT applications
- High-traffic systems
- Supply chain management systems

Introduction to Microservices Architecture

Microservices architecture encompasses a conglomeration of loosely coupled components that can be built using a collection of homogenous or heterogenous technologies. Microservices can be used to build, deploy, and scale components individually and independently of each other. This architectural approach represents a great leap forward in software development by providing organizations with the requisite agility and flexibility to operate efficiently in today's digital world. Because microservices architecture is a scalable architecture, it provides enterprises with the ability to scale existing services as needed.

There's a plethora of benefits of microservices architecture, such as the following:

- Fault tolerance
- Modularity
- Improved scalability
- Reduced coupling
- Better ROI
- Faster releases
- Faster development

Building a Microservices-Based Application Using CQRS

When building microservices-based applications, you can take advantage of the CQRS design pattern and the MediatR library to manage the command and query responsibilities of your application efficiently. This approach fosters separation of concerns, which in turn enables you to build an application that contains scalable, efficient, and maintainable source code.

In this section, you'll build a microservices-based application using CQRS. Let's now examine how to build a simple ASP.NET Core 9 Web API application using CQRS. You'll implement a simple order processing application that demonstrates how you can use CQRS in ASP.NET Core. A typical Order Processing System is composed of several entities, such as Supplier, Order, Product, Customer, etc. For the sake of simplicity and brevity, you'll build the Product module of the application in this example.

In the next section, let's examine how to create an ASP.NET Core 9 project in Visual Studio 2022.

Create a New ASP.NET Core 9 Project in Visual Studio 2022

You can create a project in Visual Studio 2022 in several ways, such as, from the Visual Studio 2022 Developer Command Prompt or by launching the Visual Studio 2022 IDE. When you launch Visual Studio 2022, you'll see the Start window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2022 IDE.

Now that you know the basics, let's start setting up the project. To create a new ASP.NET Core 8 Project in Visual Studio 2022:

1. Start the Visual Studio 2022 IDE.
2. In the **Create a new project** window, select "ASP.NET Core Web API" and click Next to move on.
3. Specify the project name as **ShoppingCartSystem** and the path where it should be created in the **Configure your new project** window.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click Next to move on.
5. In the next screen, specify the target framework and authentication type as well. Ensure that the "Configure for HTTPS," "Enable Docker Support," "Do not use top-level statements", and the "Enable OpenAPI support" checkboxes are unchecked because you won't use any of these in this example.
6. Remember to leave the **Use controllers** checkbox checked because you won't use minimal API in this example.
7. Click Create to complete the process.

A new ASP.NET Core Web API project is created. You'll use this project to implement the CQRS pattern in ASP.NET Core and C#.

Install Entity Framework Core

So far so good. The next step is to install the necessary NuGet Package(s) for working with Entity Framework Core and SQL Server. To install these packages into your project, right-click on the solution and the select **Manage NuGet Packages for Solution....**

Now search for the NuGet packages named Microsoft.EntityFrameworkCore and Microsoft.EntityFrameworkCore.InMemory packages in the search box and install them one after the other. Alternatively, you can type the commands shown below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package
Microsoft.EntityFrameworkCore
PM> Install-Package
   Microsoft.EntityFrameworkCore.
SqlServer
```

Alternatively, you can install these packages by executing the following commands at the Windows Shell:

```
dotnet add package
Microsoft.EntityFrameworkCore
dotnet add package
 Microsoft.EntityFrameworkCore.
SqlServer
```

Introducing the Mediator Pattern

The mediator design pattern is a behavioral pattern that helps decrease dependencies among objects and regulates how they can interact with each other effectively. This pattern prevents objects from communicating directly, instead requiring them to communicate through a mediator object. As a result, it helps build applications that are loosely coupled, and easier to manage and maintain. **Figure 4** illustrates the Mediator pattern.

Introducing MediatR

To implement the mediator design pattern, you can take advantage of the open-source library called MediatR. This library enables you to implement the CQRS with ease and manage the command and query handlers effectively. It allows easy implementation of CQRS by offering an effective way of dealing with command and query handlers. In essence, MediatR acts as a mediator that directs commands and queries to the appropriate handlers.

The key benefits of MediatR include the following:

- Promotes loose coupling
- Facilitates easy maintainability and testability
- Helps adhere to the single responsibility principle (SRP)
- Enables clear communication between objects

Figure 5 shows how MediatR works by delegating the request to the respective handlers.

Install MediatR in ASP.NET Core

You can install the MediatR library from NuGet. To do that, right-click on the solution and the select Manage NuGet Packages for Solution....

Now search for the NuGet packages named MediatR, and MediatR.Extensions.Microsoft.DependencyInjection packages in the search box and install them one after the other. Alternatively, you can write the commands given below at the NuGet Package Manager Command Prompt:

```
PM> Install-Package MediatR
PM> Install-Package
```

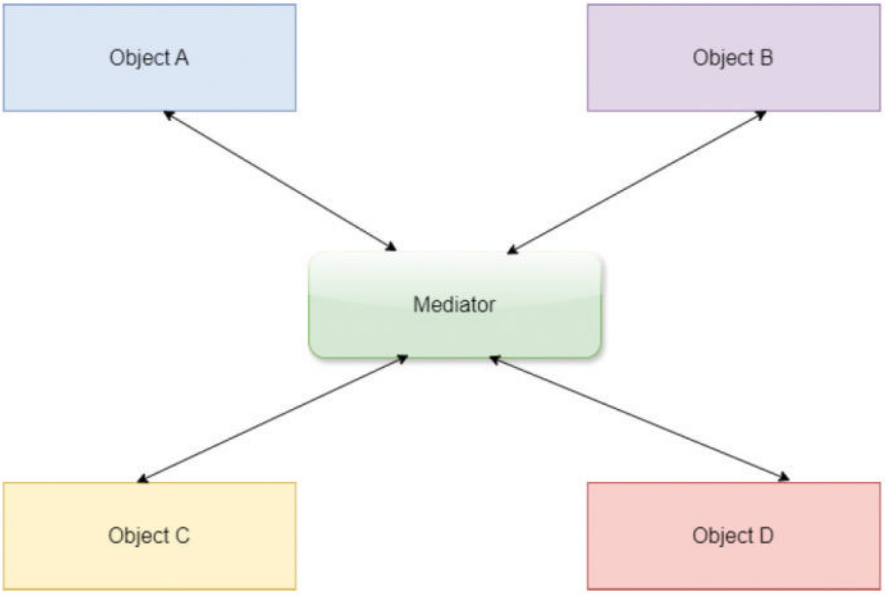


Figure 4: Demonstrating the Mediator pattern

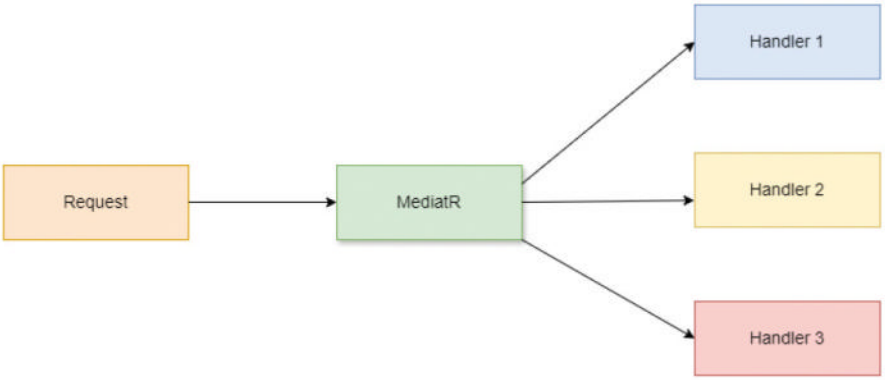


Figure 5: Demonstrating how MediatR delegates the requests to the respective handlers

```
   MediatR.Extensions.Microsoft.
DependencyInjection
```

Alternatively, you can install these packages by executing the following commands at the Windows Shell:

```
dotnet add package MediatR
dotnet add package
 MediatR.Extensions.Microsoft.
DependencyInjection
```

The MediatR.Extensions.Microsoft.DependencyInjection helps you register the MediatR handlers in your ASP.NET Core application automatically.

Register MediatR in ASP.NET Core

You should register the MediatR handlers with the services collection of your application. To do this, write the following line of code in the Program.cs file:

```
builder.Services.AddMediatR
(cfg => cfg.
RegisterServicesFromAssembly
(Assembly.
GetExecutingAssembly()));
```

Create a Request in MediatR

In MediatR, messages can be of two types. These include requests (commands/queries) and notifications (for events). You can define a request in MediatR by implementing the IRequest<TResponse> interface, where TResponse is the type of response, i.e., Order, Product, Customer, Supplier, etc. The following code snippet illustrates how you can define a request in MediatR:

```
public class GetCustomerQuery
: IRequest<Customer>
{
    public Guid CustomerId { get; set; }

    public GetCustomerQuery
        (Guid CustomerId)
    {
        this.CustomerId = CustomerId;
    }
}
```

Create a Request Handler in MediatR

You need a handler to handle the request you just created. Essentially, a handler contains the necessary logic that should be executed to handle an incoming request in MediatR. To create a handler in MediatR, create an interface that implements the IRequestHandler<TRequest, TResponse> interface as shown below:

```
public class GetCustomerQueryHandler
: IRequestHandler
<GetCustomerQuery, Customer>
{
    private readonly
    ICustomerRepository _repository;

    public GetCustomerQueryHandler
     (ICustomerRepository repository)
    {
        _repository = repository;
    }

    public async Task<User> Handle
```

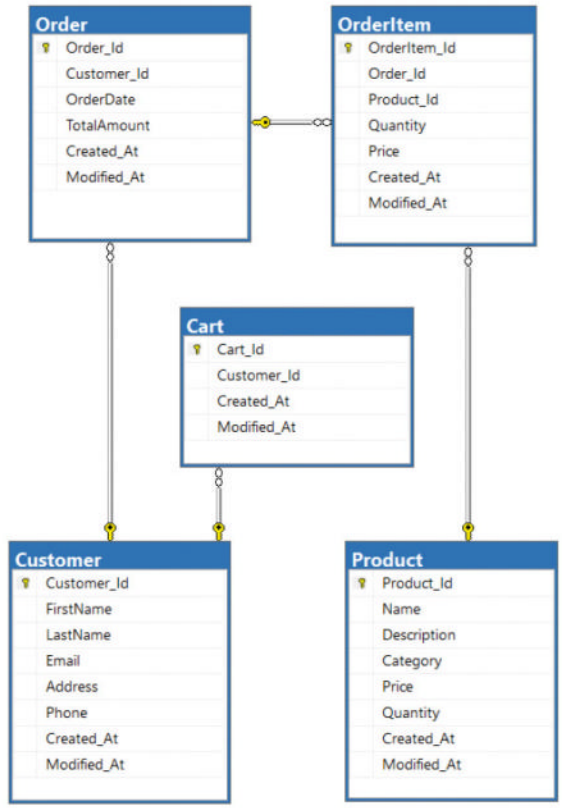


Figure 6: The database design of the ShoppingCartSystem database

Listing 1: Create the database tables

```
CREATE TABLE Product (
    [Product_Id]
    UniqueIdentifier PRIMARY KEY,
    [Name] varchar(255) NOT NULL,
    [Description] Text NOT NULL,
    [Category] varchar(50) NOT NULL,
    [Price] DECIMAL(10, 2),
    [Quantity] INT,
    [Created_At] DATETIME,
    [Modified_At] DATETIME

);

CREATE TABLE Customer (
    [Customer_Id]
    UniqueIdentifier PRIMARY KEY,
    [FirstName] VARCHAR(50),
    [LastName] VARCHAR(50),
    Email VARCHAR(100),
    [Address] VARCHAR(255),
    [Phone] VARCHAR(15),
          [Created_At] DATETIME,
    [Modified_At] DATETIME
);

CREATE TABLE [Order] (
    [Order_Id]
    UniqueIdentifier PRIMARY KEY,
    [Customer_Id] UniqueIdentifier,
    [OrderDate] TIMESTAMP,
    [TotalAmount] DECIMAL(10, 2),
    [Created_At] DATETIME,
    [Modified_At] DATETIME,
    FOREIGN KEY (Customer_Id)
    REFERENCES Customer(Customer_Id)
);

CREATE TABLE OrderItem (
    [OrderItem_Id]
    UniqueIdentifier PRIMARY KEY,
    [Order_Id] UniqueIdentifier,
    [Product_Id] UniqueIdentifier,
    [Quantity] INT,
    [Price] DECIMAL(10, 2),
    [Created_At] DATETIME,
    [Modified_At] DATETIME,
    FOREIGN KEY (Order_Id)
    REFERENCES [Order](Order_Id),
    FOREIGN KEY (Product_Id)
    REFERENCES Product(Product_Id)
);

CREATE TABLE Cart (
    [Cart_Id]
    UniqueIdentifier PRIMARY KEY,
    [Customer_Id] UniqueIdentifier,
    [Created_At] DATETIME,
    [Modified_At] DATETIME,
    FOREIGN KEY (Customer_Id)
    REFERENCES Customer(Customer_Id)
);
```

```
    (GetCustomerQuery request,
    CancellationToken token)
    {
        return await _repository.
        GetCustomerById
        (request.CustomerId);
    }
}
```

Create a Notification in MediatR

You can also create notifications and notification handlers using MediatR. Assume that you want to send notifications when a customer record is deleted from the database. To create a notification in MediatR, implement the INotification interface, as shown below.

```
public class
CustomerDeletedNotification :
INotification
{
    public Guid CustomerId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

You'll create a handler for the CustomerDeletedNotification in the next section.

Create a Notification Handler in MediatR

To create a notification handler in MediatR, implement the INotificationHandler<TNotification> interface, as shown in the code snippet given below.

```
public class
CustomerDeletedNotificationlHandler :
INotificationHandler
<CustomerDeletedNotification>
{
    public async Task Handle(
CustomerDeletedNotification
notification,
CancellationToken token)
    {
        //Write your code here
        //to send notification(s)
        //when an existing customer record
        // is deleted from the data store
    }
}
```

Now that you know how to work with MediatR, in the sections that follow, you'll implement a simple microservices-based application that leverages the MediatR library.

Create the Shopping Cart System Database

Create a new database called ShoppingCartSystem using the following script:

```
Create database ShoppingCartSystem
```

Next, create the Product, Customer, Order, OrderItem, and the Cart database tables inside the ShoppingCartSystem database using the script given in **Listing 1**.

Figure 6 demonstrates the database diagram of the ShoppingCartSystem database.

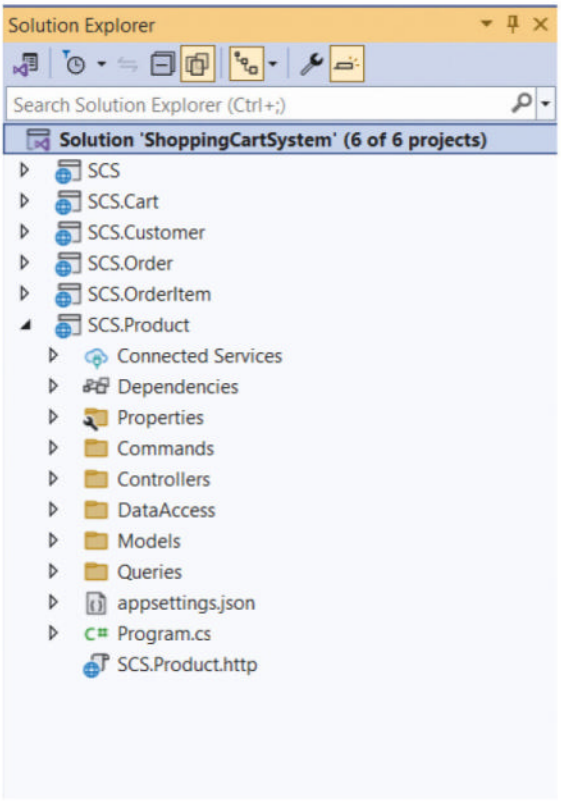


Figure 7: The Solution structure of the ShoppingCartSystem

Create the Solution Structure

As evident from the database design, the ShoppingCartSystem application is comprised of the Product, Customer, Cart, Order, and OrderItem microservices. So, you should create five WebAPI projects for each of them in the solution you created earlier. You'll also create solution folders to organize the files in each of the projects. **Figure 7** shows how the solution structure looks.

In this example, for the sake of simplicity and brevity, you'll create only the **Product** microservice. In the sections that follow, you'll create classes and interfaces pertaining to the Product microservices-based application.

Create the Product Microservice

In this example, you'll build the Product microservice application or the Product API. The product microservice application is comprised of the following files:

- **Product.cs:** This represents the product model that contains domain-specific data and (optionally) business logic.
- **IProductRepository.cs:** This represents the IProductRepository interface that contains the declaration of the operations supported by the product repository.
- **ProductRepository.cs:** This represents the product repository class that implements the members of the IProductRepository interface.
- **ProductDbContext.cs:** This represents the product data context used to perform CRUD operations for the Product table in the database.

- **GetProductByIdQuery.cs:** This represents the query for retrieving a product record based on its ID.
- **GetProductByIdQueryHandler.cs:** This represents the query handler for the GetProductByIdQuery that contains the logic for returning the product record.
- **GetAllProductsQuery.cs:** This represents the query for retrieving all product records.
- **GetAllProductsQueryHandler.cs:** This represents the query handler for the GetAllProductsQuery that contains the logic for returning all product records.
- **CreateProductCommand.cs:** This represents the required operations to create a product record in the database.
- **CreateProductCommandHandler.cs:** This represents the command handler that contains the implementation of the CreateProductCommand operation.
- **UpdateProductCommand.cs:** This represents the operations to be executed to update an existing product record in the database.
- **UpdateProductCommandHandler.cs:** This represents the command handler that contains the implementation of the UpdateProductCommand operation.
- **DeleteProductCommand.cs:** This represents the operations to be executed to delete a product record in the database.
- **DeleteProductCommandHandler.cs:** This represents the command handler that contains the implementation of the DeleteProductCommand operation.
- **appsettings.json:** This represents the application's settings file where you can configure the database connection string, logging metadata, etc.
- **Program.cs:** Any ASP.NET Core application contains a file where the startup code required by the application resides. This file is named Program.cs where the services required by your application are configured. You can specify dependency injection (DI), configuration, middleware, and much more information in this file.

Specify the Database Connection String

Your application requires a connection string to establish a connection to the database which, in turn, contains the necessary information about the database connection and any initialization parameters sent by a data provider to a data source. Typically, a connection string contains the name of the database to connect to, the instance name of the database server where the database resides, and some other settings pertaining to security of the database.

In ASP.NET Core, the application's settings are stored in a file known as appsettings.json. This file is created by default when you create a new ASP.NET Core project. You can take advantage of the ConnectionString property to retrieve or store the connection string for a database. You can specify the connection string in the appsettings.json file, as shown below:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "SCSDbSettings":
    "Write your connection string here."
  },
  "AllowedHosts": "*"
}
```

You'll use this connection string to enable the application to connect to the database in a section later in this article.

Create the Model Classes

First off, create two solution folders, one named Models and the other DataAccess. The former will contain one or more model classes, and the latter will have the data context and repository interfaces and classes. It should be noted that you can always create multiple data context classes in the same project. If your data context class contains many entity references, it's good practice to split the data context among multiple data context classes rather than having one large data context class.

Create a new class called Product in a file named Product.cs inside the Models folder and write the following code in there:

```
namespace SCS.Product.Models
{
    public record Product
    {
        public Guid Product_Id
        { get; set; }
        public string Product_Name
        { get; set; } = default!;
        public string Product_Description
        { get; set; } = default!;
        public string Product_Category
        { get; set; } = default!;
        public decimal Product_Price
        { get; set; } = default!;
        public int Product_Quantity
        { get; set; } = default!;
        public DateTime Created_At
        { get; set; } = DateTime.Now;
        public DateTime Modified_At
        { get; set; } = DateTime.Now;
    }
}
```

In this implementation, you'll use only one model class: Product.

Create the Data Context

In Entity Framework Core (EF Core), a data context is a component used by an application to interact with the database and manage database connections, and to query and persist data in the database. Let's now create the data context class to enable the application to interact with the database to perform CRUD (Create, Read, Update, and Delete) operations.

To do this, create a new class named ProductDbContext that extends the DbContext class of EF Core and write the following code in there.

```
public class ProductDbContext :
DbContext
```

every woman
in TECH FORUM

EVERYWOMAN
IN TECH FORUM

IGNITING LEADERSHIP POTENTIAL

The everywoman in Tech Forum brings together a community of women working in Tech from entry-levels through to the C-level. The Forum will enable you to hear from and connect with leading women and men in tech, giving you the opportunity to gain industry insights and share best practice and knowledge directly with peers and leaders from the Tech industry and beyond.

13 MARCH 2025

Book your delegate pass!

For group booking rates and sponsorship opportunities please speak to us via contact@everywoman.com

EVERYWOMAN.COM/TECH-FORUM

```
{
    public DbSet <Models.Product>
    Products { get; set; }

    protected override void
    OnConfiguring
    (DbContextOptionsBuilder
    optionsBuilder)
    {
        base.OnConfiguring
        (optionsBuilder);
    }
}
```

In the preceding piece of code, the statement base.OnConfiguring(optionsBuilder) calls the OnConfiguring method of the base class of your ProductDbContext. Because the base class of the ProductDbContext class is DbContext, the call does nothing in particular.

You can specify your database connection string in the OnConfiguring overloaded method of the ProductDbContext class. However, in this implementation, you'll store the database connection settings in the AppSettings.json file and read it in the Program.cs file to establish a database connection.

Note that your custom data context class (the ProductDbContext class in this example), must expose a public constructor that accepts an instance of type DbContextOptions <ApplicationDbContext> as an argument. This is needed to enable the runtime to pass the context configuration using a call to the AddDbContext() method to your custom DbContext class. The following code snippet illustrates how you can define a public constructor for your data context class.

```
public ProductDbContext
(DbContextOptions
<ProductDbContext> options,
IConfiguration configuration) :
base(options)
{
   _configuration = configuration;
}
```

Seed the Database

You might often want to work with data seeding when using Entity Framework Core (EF Core) to populate a blank database with an initial or minimal data set. Data seeding is a one-time process of loading data into a database. The EF Core framework provides an easy way to seed the data using the OnModelCreating() method of the DbContext class.

To generate fake data in your ASP.NET Core application, you can take advantage of the Bogus open-source library. It helps you to seed your database by taking advantage of randomly generated but realistic data. To use this library, you should install the Bogus (https://www.nuget.org/packages/bogus) library from NuGet into your project.

The following code snippet illustrates how you can generate data using random data from the Bogus library:

```
private Models.Product[]
GenerateProductData()
{
      var productFaker = new Faker <
      SCS.Product.Models.Product> ()
      .RuleFor(x => x.Product_Id,
       f => Guid.NewGuid())
      .RuleFor(x =>
       x.Product_Name, f =>
       f.Commerce.ProductName())
       .RuleFor(x => x.Product_Description,
       f => f.Commerce.ProductDescription())
       .RuleFor(x => x.Product_Category,
       f => f.Commerce.ProductMaterial())
       .RuleFor(x => x.Product_Price,
```

Listing 2: The ProductDbContext class

```
using Bogus;
using Microsoft.EntityFrameworkCore;

namespace SCS.Product.DataAccess
{
    public class ProductDbContext : DbContext
    {
        private readonly IConfiguration _configuration;
        public ProductDbContext
        (DbContextOptions<ProductDbContext> options,
            IConfiguration configuration) : base(options)
        {
            _configuration = configuration;
        }

        protected override void OnConfiguring
       (DbContextOptionsBuilder optionsBuilder)
        {
            _ = optionsBuilder.UseSqlServer(_configuration.
                GetConnectionString("SCSDbSettings")).
                EnableSensitiveDataLogging();
        }

        public DbSet<Models.Product>
        Products { get; set; }

        protected override void OnModelCreating
        (ModelBuilder modelBuilder)
        {
            modelBuilder.Entity< Models.Product>().
            ToTable("Product");
            modelBuilder.Entity< Models.Product>().
            HasKey (p => p.Product_Id);
            var products = GenerateProductData();
            modelBuilder.Entity<Models.Product>().
            HasData(products);
        }

        private Models.Product[] GenerateProductData()
        {
            var productFaker = new
            Faker<SCS.Product.Models.Product>()
            .RuleFor(x => x.Product_Id,
            f => Guid.NewGuid())
            .RuleFor(x => x.Product_Name,
            f => f.Commerce.ProductName())
            .RuleFor(x => x.Product_Description,
            f => f.Commerce.ProductDescription())
            .RuleFor(x => x.Product_Category,
            f => f.Commerce.ProductMaterial())
            .RuleFor(x => x.Product_Price,
            f => Math.Round(f.Random.Decimal
            (1000, 5000), 2));
            return productFaker.Generate
            (count: 5).ToArray();
        }
    }
}
```

```
        f => Math.Round(f.Random.Decimal
        (1000, 5000), 2));
    return productFaker.
    Generate(count: 5).ToArray();
}
```

Invoke the GenerateProductData method in the OnModelCreating method to populate the database with randomly generated data, as shown in the following piece of code:

```
protected override
void OnModelCreating
(ModelBuilder modelBuilder)
{
    modelBuilder.Entity
    <Models.Product> ().
    ToTable("Product");
    modelBuilder.Entity
    <Models.Product> ().
    HasKey(p => p.Product_Id);

    var products =
    GenerateProductData();
    modelBuilder.Entity
    <Models.Product> ().
    HasData(products);
}
```

The complete source code of the ProductDbContext class is given in **Listing 2**.

Register the ProductDb data context instance as a service with the services container of ASP.NET Core using the following piece of code in the Program.cs file.

```
builder.Services.AddDbContext
<ProductDbContext>(options =>
options.UseSqlServer(@"Data Source=
<<Specify the data source here>>;
Initial Catalog=
<<Specify the initial catalog here>>;
Trusted_Connection=True;
TrustServerCertificate=True;
Integrated Security=True;"));
```

If your application needs to perform multiple units of work, it's advisable to use a DbContext factory instead. To do this, register a factory by calling the AddDbContextFactory method in the Program.cs file of your project, as shown in the following code example:

```
builder.Services.AddDbContextFactory
<ProductDbContext>(options =>
{
    options.UseSqlServer
(builder.Configuration
["ConnectionStrings:SCSDbSettings"]);
});
```

Query Product Data from the Database

In this example, you'll create the following commands and queries:

- Commands
  - Create
  - Update
  - Delete
- Queries
  - Get
  - GetAll

Note that any command class should extend the IRequest<T> interface pertaining to the MediatR library.

> The DbContext class in the EF Core library is not thread safe. Refrain from sharing data context between threads.

The GetProductById Query

To read data from the database, take advantage of queries and query handers. In this example, you'll implement two types of queries and query handlers, one of them to read a product record by ID and the other to retrieve a list of all product records from the data store.

The following code snippet illustrates how you can define a query named GetProductByIdQuery inside the /Queries solution folder of the project to retrieve a product record from the database based on the product ID.

```
using MediatR;

namespace SCS.Product.Queries
{
    public record GetProductByIdQuery :
    IRequest<Models.Product>
    {
        public Guid Id { get; set; }
    }
}
```

Next, create a new .cs file named GetProductByIdHandler.cs inside the /Queries solution folder that contains a class named GetProductByIdQueryHandler that, in turn, encapsulates the logic required to retrieve a product record from the data store, as shown in **Listing 3**.

The GetAllProductsQuery

Next, create a file named GetAllProductsQuery in a file having the same name with a .cs extension inside the /Queries folder and write the following code in there:

```
using MediatR;

namespace SCS.Product.Queries
{
    public record GetAllProductsQuery :
    IRequest<List<Models.Product>>;
}
```

The GetAllProductsQueryHandler class encapsulates the logic for retrieving a list of all product records from the database, as shown in **Listing 4**.

Create, Update, and Delete Products

Now that you know how to query data from the Product database table, let's understand how you can create a new product record, update an existing product record, and delete a product record from the database. To do this, you need to create commands to handle each of the Create, Update, and Delete operations.

Create the CreateProductCommand

Create a new .cs file named CreateProductCommand.cs inside the /Commands folder of the project and replace the default generated source code with the code snippet given below:

```
using MediatR;

namespace SCS.Product.Commands
{
    public record CreateProductCommand :
    IRequest<Models.Product>
    {
        public Guid Id { get; set; }
        public string Product_Name
        { get; set; } = default!;
        public string Product_Description
        { get; set; } = default!;
        public string Product_Category
        { get; set; } = default!;
        public decimal Product_Price
        { get; set; } = default!;
        public DateTime Created_At
        { get; set; } = DateTime.Now;
        public DateTime Modified_At
        { get; set; } = DateTime.Now;
    }
}
```

The CreateProductCommand should have a corresponding handler to persist the product data into the data store. To do this, create another .cs file named CreateProductCommandHandler.cs and replace the default generated source code with the code given in **Listing 5**.

Code Explanation

The following series of steps explain how the CreateProductHandler works:

- The CreateProductCommandHandler contains an asynchronous method named Handle that accepts an instance of CreateProductCommand and a CancellationToken object as a parameter.
- Inside the Handle method, a new Product object is created.
- This new object is populated with the data retrieved from the CreateProductCommand instance passed to the Handle method as a parameter.
- The new Product instance is finally saved into the database by making a call to the SaveChangesAsync method on the data context instance.
- The product ID of the newly created record is then returned.

Create the UpdateProductCommand

Now, create a command named UpdateProductCommand under the /Commands solution folder to update a product record in the database, as shown in the code snippet given below:

```
using MediatR;

namespace SCS.Product.Commands
{
  public record UpdateProductCommand :
  IRequest<Product>
  {
     public Guid Product_Id
     { get; set; }
     public string
     Product_Name
     { get; set; } = default!;
     public string
     Product_Description
     { get; set; } = default!;
     public string
     Product_Category
```

Listing 3: The GetProductByIdQueryHandler class

```
using MediatR;
using SCS.Product.DataAccess;

namespace SCS.Product.Queries
{
    public class GetProductByIdQueryHandler :
    IRequestHandler<GetProductByIdQuery,
    Models.Product>
    {
        private readonly
        IProductRepository _repository;

        public GetProductByIdQueryHandler
        (IProductRepository repository)
        {
            _repository = repository;
        }

        public async Task<Models.Product>
        Handle(GetProductByIdQuery productRequest,
        CancellationToken cancellationToken)
        {
            return await
            _repository.GetByIdAsync
            (productRequest.Id);
        }
    }
}
```

Listing 4: The GetAllProductsQueryHandler

```
using MediatR;
using SCS.Product.DataAccess;

namespace SCS.Product.Queries
{
    public class GetAllProductsQueryHandler :
    IRequestHandler<GetAllProductsQuery,
    IEnumerable<Models.Product>>
    {
        private readonly IProductRepository _repository;

        public GetAllProductsQueryHandler
        (IProductRepository repository)
        {
            _repository = repository;
        }

        public async Task<IEnumerable<Models.Product>>
        Handle(GetAllProductsQuery request,
        CancellationToken cancellationToken)
        => await _repository.GetAllAsync();
    }
}
```

```
    { get; set; } = default!;
    public decimal Product_Price
    { get; set; } = default!;
    public DateTime Created_At
    { get; set; } = DateTime.Now;
    public DateTime Modified_At
    { get; set; } = DateTime.Now;
  }
}
```

The UpdateProductCommand should have a corresponding handler to update a product record based on the product ID. To do this, create another .cs file named UpdateProductCommandHandler.cs and write the source code given in **Listing 6** in there.

Code Explanation

The following series of steps explain how the UpdateProductHandler works:

- The UpdateProductCommandHandler contains an asynchronous method named Handle that accepts an instance of UpdateProductCommand and a CancellationToken object as a parameter.
- Inside the Handle method, the Product record based on its ID is retrieved from the database.
- This Product object is updated with the data retrieved from the UpdateProductCommand instance passed to the Handle method as a parameter.
- The updated Product instance is finally saved into the database by making a call to the SaveChangesAsync method on the data context instance.
- The Product instance is then returned.

Create the DeleteProductCommand

Next, create another command named DeleteProductCommand under the /Commands solution folder to delete a product record in the database, as shown in the code snippet given below.

```
using MediatR;

namespace SCS.Product.Commands
{
    public record
    DeleteProductCommand(Guid Id) :
    IRequest;
}
```

The DeleteProductCommand should also have a corresponding handler to delete a product record based on the product ID. To do this, create another .cs file named DeleteProductCommandHandler.cs and write the source code given in **Listing 7** in there.

Code Explanation

The following series of steps explain how the DeleteProductHandler works:

- The DeleteProductCommandHandler contains an asynchronous method named Handle that accepts an instance of DeleteProductCommand and a CancellationToken object as a parameter.
- Inside the Handle method, the Product record to be deleted based on its ID is retrieved from the database.

Listing 5: The CreateProductCommandHandler class

```
using MediatR;
using SCS.Product.DataAccess;

namespace SCS.Product.Commands
{
   public class CreateProductCommandHandler
    (IProductRepository productRepository) :
    IRequestHandler<CreateProductCommand,
     Models.Product>
    {
      public async Task<Models.Product>
      Handle(CreateProductCommand productCommand,
      CancellationToken cancellationToken)
      {
        var product = new Models.Product
        {
            Product_Name = productCommand.Product_Name,
            Product_Description =
            productCommand.Product_Description,
            Product_Category =
            productCommand.Product_Category,
            Product_Price = productCommand.Product_Price,
            Created_At = DateTime.UtcNow,
            Modified_At = DateTime.UtcNow
        };

       await productRepository.CreateAsync(product);
       return product;
      }
   }
}
```

Listing 6: The UpdateProductCommandHandler class

```
using MediatR;
using SCS.Product.DataAccess;

namespace SCS.Product.Commands
{
    public class UpdateProductCommandHandler
    (IProductRepository productRepository) :
        IRequestHandler<UpdateProductCommand,
        Models.Product>
    {
        public async Task<Models.Product> Handle
        (UpdateProductCommand productRequest,
        CancellationToken cancellationToken)
        {
            var product = await
            productRepository.GetByIdAsync
            (productRequest.Id);

            if (product != null)
            {
                product.Product_Name =
                productRequest.Product_Name;
                product.Product_Description =
                productRequest.Product_Description;
                product.Product_Category =
                productRequest.Product_Category;
                product.Product_Price =
                productRequest.Product_Price;
                product.Modified_At = DateTime.UtcNow;
                await productRepository.UpdateAsync(product);
                return product;
            }

            return default;
        }
    }
}
```

- The Product instance is finally deleted by making a call to the SaveChangesAsync method on the data context instance.
- An appropriate exception message is thrown if the ProductId pertaining to the product record to be deleted doesn't exist in the database.

Create the ProductRepository Class

A repository class is an implementation of the Repository design pattern and is one that manages data access. The application takes advantage of the repository instance to perform CRUD operations against the database. Now, create a new class named ProductRepository in a file having the same name with a .cs extension. Then write the following code in there:

```
public class ProductRepository :
IProductRepository
{

}
```

> In the Product model class, you'll observe the usage of Models.Product when referring to the Product class. This is needed because the names of the project and the model class are identical. You can avoid this by using different names anyway.

The ProductRepository class, illustrated in the code snippet below, implements the methods of the IProductRepository interface. Here is how the IProductRepository interface should look:

```
namespace SCS.Product.DataAccess
{
    public interface IProductRepository
    {
        public Task<Ienumerable
        <Models.Product>> GetAllAsync();

        public Task<Models.Product>
         GetByIdAsync(Guid id);

        public Task CreateAsync
        (Models.Product product);

        public Task UpdateAsync
        (Models.Product product);

        public Task DeleteAsync
        (Models.Product product);
    }
}
```

The complete source code of the ProductRepository class is given in **Listing 8**.

Create the ProductController Class

Now, create a new controller named ProductController in the Controllers folder of the project. The following code snippet shows how you can take advantage of constructor injection to pass an instance of the query handler using the constructor and then use it to retrieve all product records from the database.

```
private readonly IRequestHandler
    <GetAllProductsQuery,
    IEnumerable
    <Models.Product>>
    _getAllProductQueryHandler;

public ProductController
(IRequestHandler
<GetAllProductsQuery, IEnumerable
<Models.Product>>
getAllProductsQueryHandler)
{
    _getAllProductQueryHandler =
    getAllProductsQueryHandler;
}

[HttpGet("GetAllProducts")]
public async Task <IEnumerable
<Models.Product>> GetAllProducts()
{
    return await
    _getAllProductQueryHandler.Handle
    (new GetAllProductsQuery(),
    new CancellationToken());
}
```

The following code snippet shows the action methods for creating, updating, and deleting product records. The first code snippet shows the CreateProduct action method that creates a new product record in the database.

```
[HttpPost(nameof(CreateProduct))]
public async Task<IActionResult>
CreateProduct(CreateProductCommand command)
{
  try
  {
```

Listing 7: The DeleteProductCommandHandler class

```
using MediatR;
using SCS.Product.DataAccess;

namespace SCS.Product.Commands
{
    public class DeleteProductCommandHandler
    (IProductRepository productRepository) :
        IRequestHandler<DeleteProductCommand>
    {
        public async Task Handle
        (DeleteProductCommand productRequest,
            CancellationToken cancellationToken)
        {
            var product = await
            productRepository.
            GetByIdAsync(productRequest.Id);

            if (product != null)
            {
                await productRepository.
                DeleteAsync(product);
            }
            else
            {
               throw new Exception
               ($"Product with Id
               {productRequest.Id} not found.");
            }
        }
    }
}
```

```
    await _createProductCommandHandler.Handle
    (command, new CancellationToken());
    return Ok
    ("Product record added successfully");
  }
  catch (Exception ex)
  {
    return StatusCode
    (StatusCodes.Status500InternalServerError,
    $"Error adding product: {ex.Message}");
  }
}
```

Similarly, the UpdateProduct action method shown in the code example below is used to update or alter an existing product record in the database.

```
[HttpPut(nameof(UpdateProduct))]
public async Task<IActionResult>
UpdateProduct(UpdateProductCommand command)
{
    try
    {
        await _updateProductCommandHandler.Handle
        (command, new CancellationToken());
        return Ok("Product record updated
        successfully");
    }
    catch (Exception ex)
    {
        return StatusCode(StatusCodes.
        Status500InternalServerError,
        $"Error updating product: {ex.Message}");
    }
}
```

Finally, the DeleteProduct action method given below is responsible for deleting an existing product record in the database.

> The action verbs HttpGet, HttpPost, HttpPut, HttpDelete, and HttpPatch are specified using attributes in the action methods in ASP.NET Core to handle various types of requests. If you've not specified any action verb in an action method, the runtime will consider the request as a HttpGet request by default.

```
[HttpDelete("DeleteProduct")]
public async Task<IActionResult>
DeleteProduct(DeleteProductCommand
deleteProductCommand)
{
    try
    {
        await _deleteProductCommandHandler.Handle
        (deleteProductCommand,
```

Listing 8: The ProductRepository Class

```
using Microsoft.EntityFrameworkCore;

namespace SCS.Product.DataAccess
{
    public class ProductRepository : IProductRepository
    {
        private readonly ProductDbContext _dbContext;

        public ProductRepository
        (ProductDbContext dbContext)
        {
            _dbContext = dbContext;
            _dbContext.Database.EnsureCreated();
        }

        public async Task<Models.Product>
        GetByIdAsync(Guid id)
        {
            return await
           _dbContext.Set
            <Models.Product>().FindAsync(id);
        }

        public async Task<Ienumerable
        <Models.Product>> GetAllAsync()
        {
            return await
           _dbContext.Set
            <Models.Product>().ToListAsync();
        }

        public async Task CreateAsync
        (Models.Product entity)
        {
            await _dbContext.Set<Models.Product>().
            AddAsync(entity);
            await _dbContext.SaveChangesAsync();
        }

        public async Task UpdateAsync
        (Models.Product entity)
        {
            _dbContext.Set<Models.Product>().
            Update(entity);
            await _dbContext.SaveChangesAsync();
        }

        public async Task DeleteAsync(Models.Product entity)
        {
            _dbContext.Set<Models.Product>().Remove(entity);
            await _dbContext.SaveChangesAsync();
        }
    }
}
```

```
        new CancellationToken());
        return Ok
        ("Product record deleted successfully");
    }
    catch (Exception ex)
    {
        return StatusCode
        (StatusCodes.Status500InternalServerError,
        $"Error deleting product: {ex.Message}");
    }
}
```

Listing 9 shows the complete source of the ProductController class.

Register the Service Instances with IServiceCollection

The following code snippet illustrates how you can register the IRequestHandler instances added as a transient service to the IServiceCollection.

SPONSORED SIDEBAR

CODE Is Hiring!

CODE Staffing is accepting resumes for various open positions ranging from junior to senior roles. We have **multiple openings** and will consider candidates who seek full-time employment or contracting opportunities.

For more information:
www.codestaffing.com.

```
builder.Services.AddScoped
<IProductRepository,
ProductRepository>();
builder.Services.AddTransient
```

```
<IRequestHandler
<GetProductByIdQuery,
Models.Product>> ();
builder.Services.AddTransient
```

Listing 9: The ProductController class

```
using MediatR;
using Microsoft.AspNetCore.Mvc;
using SCS.Product.Commands;
using SCS.Product.Queries;

namespace SCS.Product.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductController: Controller
    {
        private readonly IRequestHandler
        <GetProductByIdQuery,   Models.Product>
        _getProductByIdQueryHandler;
        private readonly IRequestHandler
        <GetAllProductsQuery  , IEnumerable
        <Models.Product>>
        _getAllProductsQueryHandler;

        private readonly IRequestHandler
        <CreateProductCommand  , Models.Product>
        _createProductCommandHandler;
        private readonly IRequestHandler
        <UpdateProductCommand  , Models.Product>
       _updateProductCommandHandler;
        private readonly IRequestHandler
        <DeleteProductCommand>
        _deleteProductCommandHandler;

        public ProductController(
            IRequestHandler <GetProductByIdQuery,

            Models.Product>
            getProductByIdQueryHandler,
            IRequestHandler <CreateProductCommand,

           Models.Product>
            createProductCommandHandler,
            IRequestHandler
            <GetAllProductsQuery,

          IEnumerable <Models.Product>>
            getAllProductsQueryHandler,
                        IRequestHandler
            <UpdateProductCommand,
            Models.Product>
            updateProductCommandHandler,
                        IRequestHandler
            <DeleteProductCommand >
            deleteProductCommandHandler)
        {
            _getProductByIdQueryHandler = =
              getProductByIdQueryHandler;
            _getAllProductsQueryHandler = =
               getAllProductsQueryHandler;

            _createProductCommandHandler = =
               createProductCommandHandler;
            _updateProductCommandHandler = =
                updateProductCommandHandler;
            _deleteProductCommandHandler = =
               deleteProductCommandHandler;
        }

        [HttpGet("GetProductById")]
        public async Task <Models.Product>
        GetProductById
        (GetProductByIdQuery getProductByIdQuery)
        {
            return await
            _getProductByIdQueryHandler.Handle
             (getProductByIdQuery,
            new CancellationToken());
        }

        [HttpGet(nameof(GetAllProducts))]
        public async Task <IEnumerable
        <Models.Product>>
        GetAllProducts()
        {
            return await
            _getAllProductsQueryHandler.
            Handle
            (new GetAllProductsQuery(),
            new CancellationToken());
        }

        [HttpPost(nameof(CreateProduct))]
        public async Task <IActionResult>
        CreateProduct(CreateProductCommand command)
        {
            try
            {
                await
                _createProductCommandHandler.
                Handle
                (command,
                new CancellationToken());
                return Ok
                ("Product record added successfully");
            }
            catch (Exception ex)
            {
                return StatusCode(StatusCodes.
                Status500InternalServerError, $
                "Error adding
                product: {ex.Message}");
            }
        }

        [HttpPut(nameof(UpdateProduct))]
        public async Task <IActionResult>
        UpdateProduct(UpdateProductCommand command)
        {
            try
            {
                await _updateProductCommandHandler.
               Handle(command,
                new CancellationToken());
                return Ok
            ("Product record updated successfully");
            }
            catch (Exception ex)
            {
                return StatusCode(StatusCodes.
                Status500InternalServerError,
                 $ "Error updating
                product: {ex.Message}");
            }
        }

        [HttpDelete("DeleteProduct")]
        public async Task <IActionResult>
        DeleteProduct(DeleteProductCommand
        deleteProductCommand)
        {
            try
            {
                 await_deleteProductCommandHandler.Handle
                 (deleteProductCommand,
                  new CancellationToken());
                 return Ok
                 ("Product record deleted successfully");
            }
            catch (Exception ex)
            {
                return StatusCode(StatusCodes.
                Status500InternalServerError,
                 $ "Error deleting
                product: {ex.Message}");
            }
        }
    }
}
```

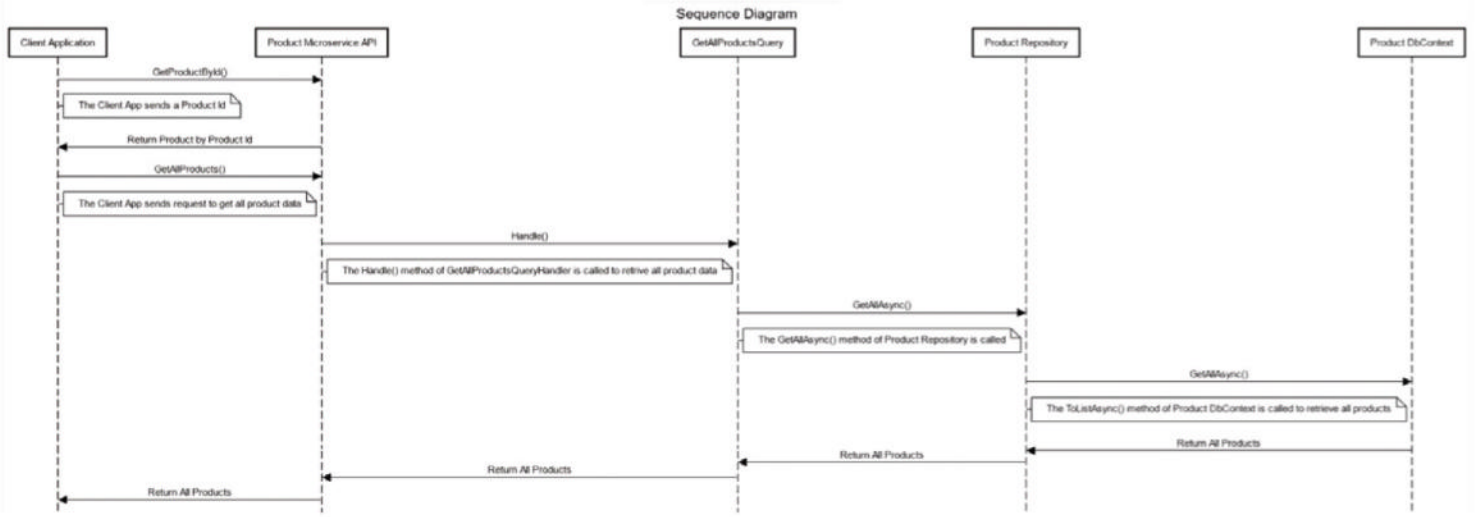

Sequence Diagram
Client Application
Product Microservice API
GetAllProductsQuery
Product Repository
Product DbContext
GetProductById()
The Client App sends a Product Id
Return Product by Product Id
GetAllProducts()
The Client App sends request to get all product data
Handle()
The Handle() method of GetAllProductsQueryHandler is called to retrive all product data
GetAllAsync()
The GetAllAsync() method of Product Repository is called
GetAllAsync()
The ToListAsync() method of Product DbContext is called to retrieve all products
Return All Products
Return All Products
Return All Products

Figure 8: The sequence diagram of the GetAllProducts HttpGet flow

```
<IRequestHandler
<GetAllProductsQuery,
IEnumerable<Models.Product>>,
GetAllProductsQueryHandler>();
builder.Services.AddTransient
<IRequestHandler
<CreateProductCommand,
Models.Product>,
CreateProductCommandHandler>();
builder.Services.AddTransient
<IRequestHandler
<UpdateProductCommand,
Models.Product>,
UpdateProductCommandHandler>();
builder.Services.AddTransient
<IRequestHandler
<DeleteProductCommand>,
DeleteProductCommandHandler>();
```

In addition, register the IproductRepository instance with the service collection, as shown below:

```
builder.Services.AddScoped
<IProductRepository, ProductRepository>();
```

The complete source code of the Program.cs file is given in **Listing 10**.

Sequence Diagram of the GetAllProducts Flow

The sequence diagram of the GetAllProducts HttpGet flow is shown in **Figure 8**:

Execute the Application

Finally, run the application and launch the popular API tool Postman. **Figure 9** shows the output upon execution of the getproductbyid endpoint.

When you run the createproduct endpoint and specify the details of the new product to be added to the database in the body of the request, a new product record is added to the Product table and the text "Product record added successfully" is returned as part of the response, as shown in **Figure 10**.

Listing 10: The Complete Program.cs file

```
global using Models = SCS.Product.Models;
using MediatR;
using Microsoft.EntityFrameworkCore;
using SCS.Product.Commands;
using SCS.Product.DataAccess;
using SCS.Product.Queries;
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddMediatR
(cfg => cfg.RegisterServicesFromAssembly
(Assembly.GetExecutingAssembly()));

builder.Services.AddScoped
<IProductRepository, ProductRepository>();
builder.Services.AddControllers();

builder.Services.AddDbContext
<ProductDbContext>(options =>
{
options.UseSqlServer
(builder.Configuration
["ConnectionStrings:SCSDbSettings"]);
});

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Best Practices

Here are some of the best practices to follow when working with CQRS:

- Define distinct Commands and Queries.
- Commands in a CQRS implementation should be task-based (i.e., CreateOrder, UpdateUser) and not data-centric.

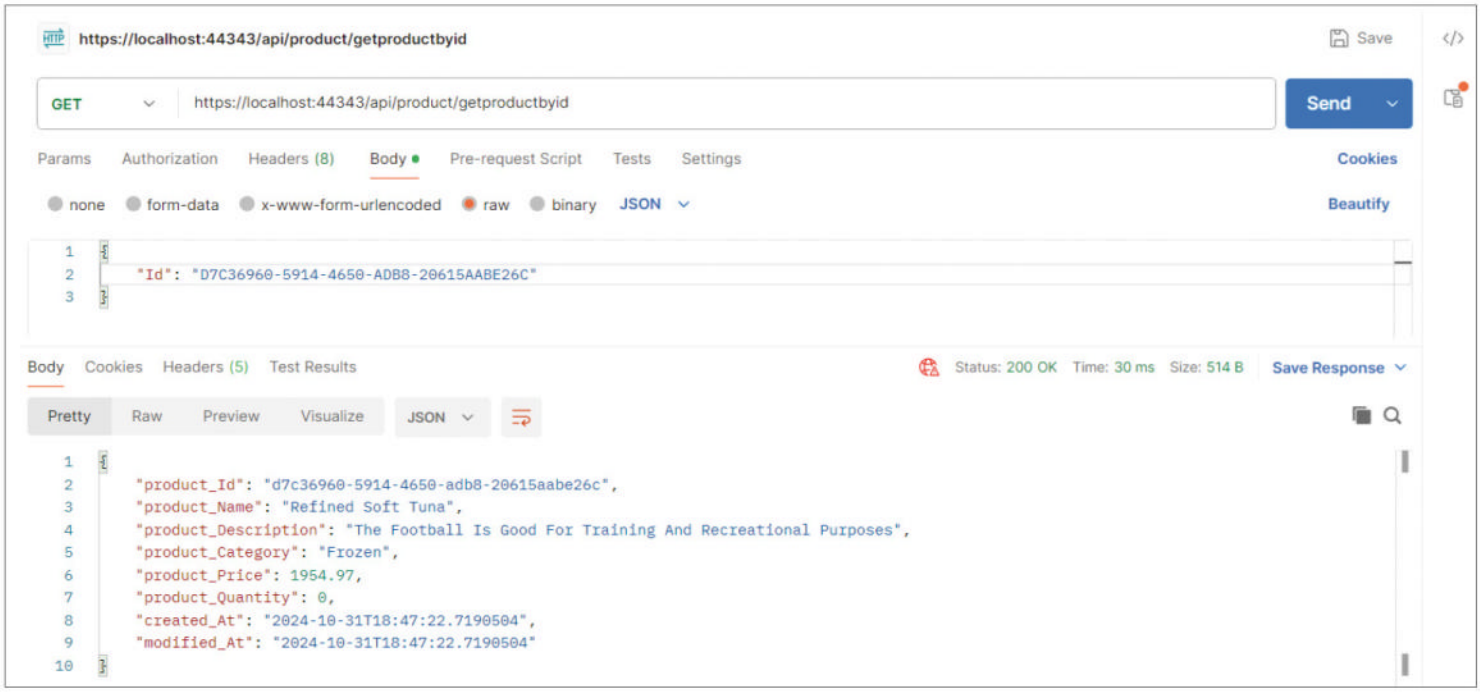


Figure 9: The getproductbyid endpoint in action

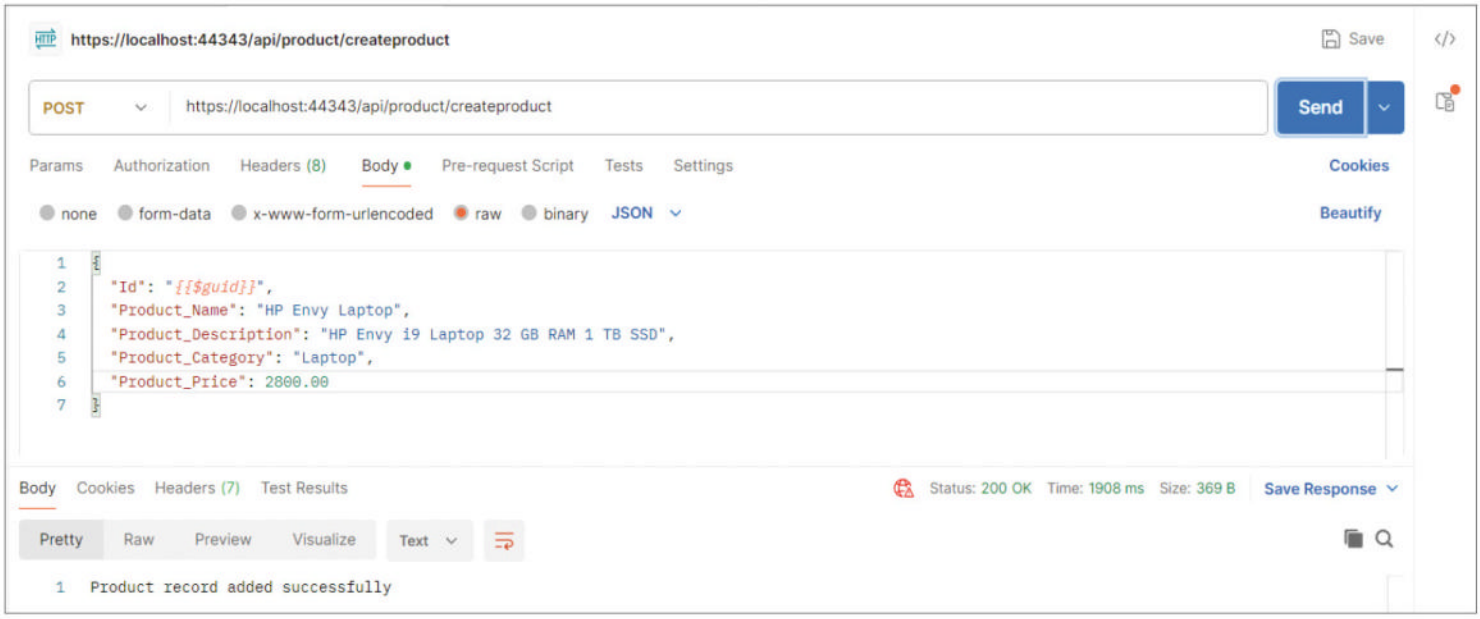


Figure 10: The createproduct endpoint in action

- Queries in a typical CQRS implementation should only return data without changing the state.
- Your write model should be normalized to ensure data integrity and consistency.
- Take advantage of MediatR for mediating commands and queries.
- You can leverage AutoMapper to transfer domain models to data transfer objects.

Where Do We Go from Here?

This article taught you about the CQRS design pattern, its benefits, components, challenges, and how to use it in microservices-based applications. You also learned how to take advantage of this design pattern to implement microservices-based applications that are scalable, efficient, and maintainable. By splitting the application's CRUD operations into two sections, namely, the command and query sides, the CQRS pattern promotes flexibility in designs, enhances security, improves performance, and helps scale your application easily. I'll discuss more design patterns related to microservices architecture in future articles.

Joydip Kanjilal

GEO WEEK

FEBRUARY 10-12, 2025
DENVER, CO - USA
geo-week.com

The intersection of
geospatial +
the **built world**

Accomplish a year's worth of geospatial business in just one week by attending Geo Week 2025

Geo Week is the premier event for the built world and geospatial professionals, created in response to the increasing convergence of technology in today's world. Geo Week's conference program and tradeshow floor feature commercial applications of 3D technologies, innovations, and case studies in the built environment, advanced airborne and terrestrial remote sensing solutions, smart products for an entire project team, and much more! Experience the future at Geo Week.

REGISTER AT GEO-WEEK.COM
Use code **SAVE100** for $100 off a conference pass or a **FREE** exhibit hall pass.

INDUSTRIES SERVED

- Architecture, Engineering & Construction
- Asset & Facility Management
- Disaster & Emergency Response
- Earth Observation & Satellite Applications
- Energy & Utilities
- Infrastructure & Manufacturing
- Land & Natural Resource Management
- Mining & Aggregates
- Surveying & Mapping
- Urban Planning & Smart Cities

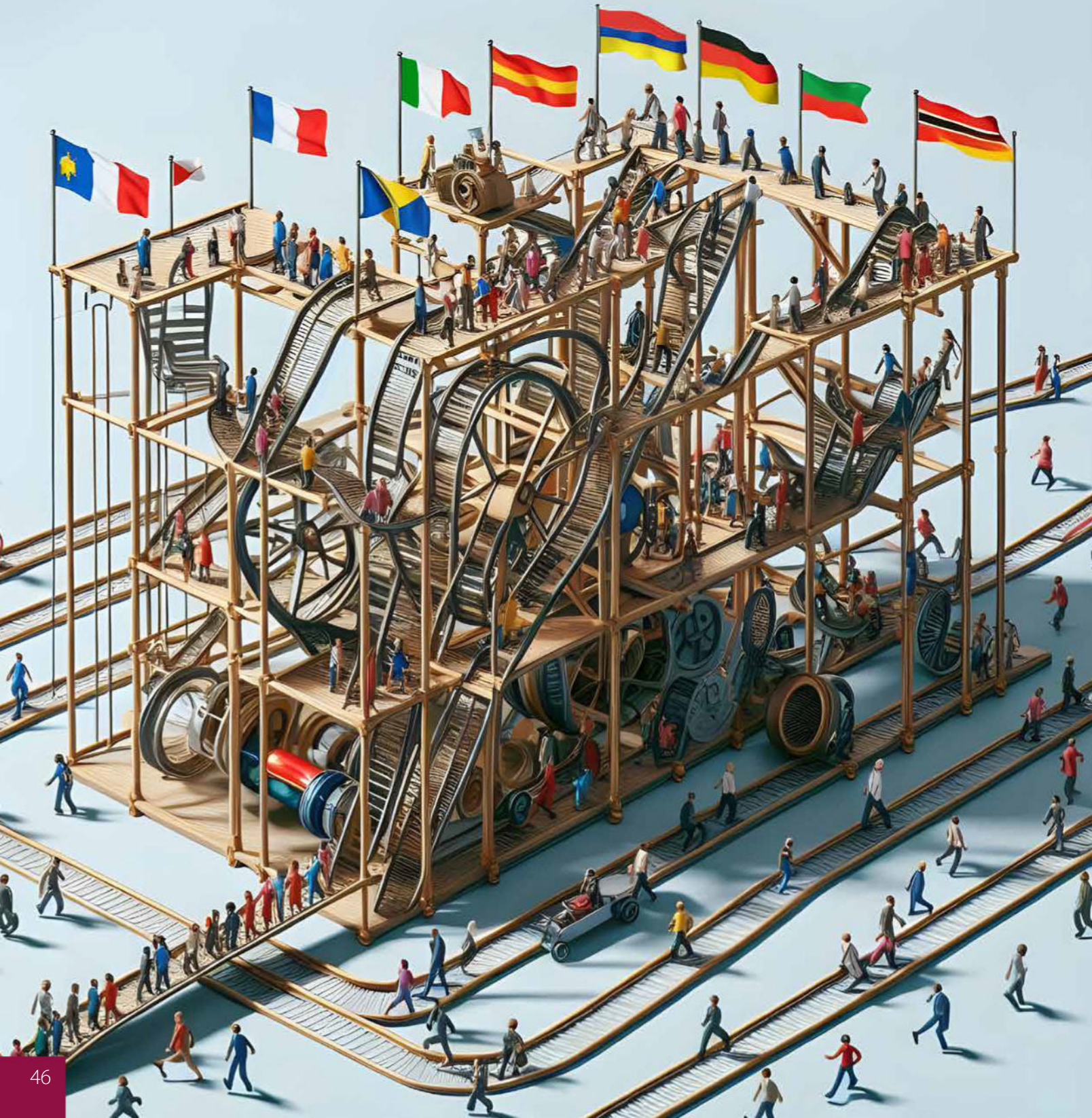
PRESENTED BY: GEO WEEK NEWS

CONFERENCE AND EVENT PARTNERS: asprs THE IMAGING & GEOSPATIAL INFORMATION SOCIETY, RCN

EVENT PARTNERS: asprs, isprs, MAPPS, RCN, USIBD, USGIF 20th ANNIVERSARY, World Geospatial Industry Council

Produced by diversified COMMUNICATIONS

Exploring LangChain: A Practical Approach to Language Models and Retrieval-Augmented Generation (RAG)



LangChain is a powerful framework for building applications that incorporate large language models (LLMs). It simplifies the process of embedding LLMs into complex workflows, enabling the creation of conversational agents, knowledge retrieval systems, automated pipelines, and other AI-driven applications. At its core, LangChain follows a modular design that allows developers to build "chains," or sequences of actions, with customizable components like prompt templates, model settings, response parsing, and memory management. It also supports integration with external data sources such as document databases, search indices, APIs, and more—a feature commonly referred to as Retrieval-Augmented Generation (RAG). This flexibility empowers developers to create tailored solutions for diverse tasks, from customer support bots to data analysis tools that extract insights from extensive datasets.

With a growing suite of tools and agents, LangChain is a top choice for developers aiming to leverage LLMs for dynamic, data-driven, and interactive applications. It offers extensive customization options and is actively maintained to support the latest advancements in LLMs and AI.

In this article, I'll guide you through the basics of using LangChain and its components. You'll learn how to combine different modules to create functional applications, including a RAG application for querying private documents using LLMs.

A Basic LangChain Example

The easiest way to get started with LangChain is to begin with a simple example. First, let's install the following libraries using the **pip** command:

```
!pip install langchain
!pip install langchain-openai
```

For this example, you'll be using LLMs from OpenAI, so you need to apply for an OpenAI API key and then save the API key in an environment variable:

```
import os

# replace with your own API Key
os.environ['OPENAI_API_KEY'] = "OpenAI API Key"
```

Do remember that OpenAI operates with a pay-for-use model, where costs are typically based on usage, like the number of tokens processed in language model queries. Go to https://platform.openai.com/docs/overview to sign up for an OpenAI account.

Components In LangChain

In a LangChain application, components are connected or "chained" to create complex workflows for natural language processing. Each component in the chain serves a specific purpose, like prompting the model, managing memory, or processing outputs, and they pass information to each other to enable more sophisticated applications. By chaining these components, you can build systems that not only generate responses but also retrieve information, maintain conversational context, summarize content, and much more. This modular approach allows flexibility, letting you create pipelines that can adapt to various tasks and data inputs based on the needs of your application.

In this basic example, you'll use the following components:

- **PromptTemplate**: This component helps create a structured template for the prompt. It allows you to specify placeholders that can be dynamically filled with specific inputs (like a question or context) each time the prompt is used. This makes it easier to standardize prompts while customizing them for each query.
- **ChatOpenAI**: This component interfaces with OpenAI's chat models, enabling the generation of responses based on the provided prompt and any contextual information. It acts as the core of the application, where the language model generates responses to the user's inputs.
- **StrOutputParser**: This component processes the raw output from the model into a usable format, such as extracting only the text content. It simplifies the response so that it can be easily displayed or further processed in your application.

By chaining these components, you can build a streamlined flow from prompt creation to response parsing, providing a solid foundation for more advanced LangChain applications. **Figure 1** shows how these components are chained together.

Let's create the first component: **PromptTemplate**:

```
from langchain import PromptTemplate

template = '''
```

Figure 1: Chaining all the components in a LangChain application

Wei-Meng Lee
weimenglee@learn2develop.net
http://www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experiences and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

```
Question: {question}
Answer: '''

prompt = PromptTemplate(
    template = template,
    input_variables = ['question']
)
```

Basically, the **PromptTemplate** contains a string template that specifies the structure of the prompt, containing a Question field where **{question}** acts as a placeholder. When the template is used, this placeholder will be replaced with the actual question input.

The next component to create is the **ChatOpenAI**:

```
from langchain_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4o-mini")
```

Here, you're making use of the "gpt-4o-mini" model from OpenAI.

The third component is the **StrOutputParser**:

```
from langchain_core.output_parsers \
import StrOutputParser

output_parser = StrOutputParser()
```

The **StrOutputParser** is used to handle the output from your model and parse it as a straightforward string. This is useful when working with responses that don't require complex parsing or structuring. **StrOutputParser** will ensure that the model's output is returned as a raw string.

> In LangChain, the | operator is used to combine multiple components (such as PromptTemplate, ChatOpenAI, and StrOutputParser) into a chain. This "piping" operator allows you to create a seamless workflow where the output of one component is automatically passed as the input to the next.

Chaining the Components

Finally, you can now combine your components into a single chain by linking the **PromptTemplate**, **ChatOpenAI** model, and **StrOutputParser**. This chaining approach allows a streamlined pipeline where you can input a question, have it processed through each component, and receive a parsed response.

```
# create the chain
chain = prompt | model | output_parser
```

Invoking the Chain

To use the chain to answer a question, call its **invoke()** method and pass in a dictionary containing the **question** key and setting its value to the question you're asking:

```
chain.invoke({"question": "Who is Steve Jobs?"})
```

The **invoke()** method in LangChain returns the final processed output after it flows through each component in the chain. You will see something like the following:

> Steve Jobs was an American entrepreneur, inventor, and business magnate best known as the co-founder of Apple Inc. He was born on February 24, 1955, and passed away on October 5, 2011. Jobs played a crucial role in the development of revolutionary products such as the Macintosh computer, iPod, iPhone, and iPad. He was known for his visionary approach to technology and design, as well as his emphasis on user experience. In addition to his work at Apple, Jobs was also the CEO of Pixar Animation Studios and played a significant role in the production of acclaimed films like "Toy Story." His leadership style and focus on innovation have left a lasting impact on the technology industry and popular culture.

> This invoke() method simplifies the interaction with your LangChain application by handling all components in sequence and directly providing the final answer.

Maintaining Conversations with Memory

If you've used ChatGPT before, you know it can handle follow-up questions seamlessly. For example, after asking, "Who is Steve Jobs?" you might follow up with, "What are some of the companies he founded?" ChatGPT understands that "he" refers to Steve Jobs and can provide relevant information about the companies he founded. This ability to maintain context across questions is possible because ChatGPT uses memory to keep track of the conversation.

The LangChain application that you built earlier, however, doesn't have memory of the previous conversation. To prove that, ask a follow-up question:

```
chain.invoke({"question":
              "What company did he found?"})
```

And you will get the following response from the model:

> Could you please provide more context or specify who you are referring to? This will help me give you a more accurate answer.

To maintain a conversation with the model, you can use the **ConversationBufferMemory** component in LangChain. The **ConversationBufferMemory** component helps store the ongoing conversation's context, allowing the model to remember previous inputs and responses. This memory buffer enables the model to refer back to earlier parts of the conversation, making follow-up questions and references more coherent.

To maintain a conversation effectively, you should modify the prompt template to include two placeholders: one for the conversation history and one for the current question. This structure allows the model to consider the entire context of the conversation when generating a response:

```
# Define the prompt template
template = '''
Previous conversation:
{history}
Question: {question}
Answer: '''

# Create the PromptTemplate with history
prompt = PromptTemplate(
    template = template,
    input_variables = ['history', 'question']
)
```

To store the history of the conversation, create an instance of the **ConversationBufferMemory** class:

```
# Set up conversational memory
memory = ConversationBufferMemory()
```

To retrieve the history of the conversation whenever you ask the model a question, you can use the following statement:

```
memory.load_memory_variables({})["history"]}
```

Here's how the above statement works:

- **memory.load_memory_variables({})**: This method retrieves the current memory variables stored in the **ConversationBufferMemory**. By passing an empty dictionary, you are requesting all memory variables without any filters.
- **["history"]**: This accesses the specific history variable from the retrieved memory. It provides the entire context of the conversation up to that point.

So now when you ask a question, you create a dictionary with two keys: **question** and **history**:

```
response = chain.invoke(
    {"question" : question,
     "history" :
     memory.load_memory_variables({})["history"]} )
print(response)
```

Essentially, whenever you ask a question, you are also passing back the history of the conversation to the model so that it can provide the context for the current question.

When the model returns a response, you should save the context to the **ConversationBufferMemory** instance using the **save_context()** method. This method allows you to store both the question and the answer, thereby updating the conversation history. Here's how you can do this:

```
memory.save_context(
    {"question": question},
    {"answer": response} )
```

Listing 1 shows the complete application that can maintain a conversation with the model.

Two questions were asked. The model prints out the following output:

```
Steve Jobs was an American entrepreneur, inventor, and business magnate best known for co-
```

Listing 1: Maintaining a conversation with the model

```
from langchain.memory import ConversationBufferMemory
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI
from langchain import PromptTemplate

# Define the prompt template
template = '''
Previous conversation:
{history}
Question: {question}
Answer: '''

# Create the PromptTemplate with history
prompt = PromptTemplate(
    template = template,
    input_variables = ['history', 'question']
)

# Set up conversational memory
memory = ConversationBufferMemory()

chain = prompt | \
        ChatOpenAI(model="gpt-4o-mini") | \
        StrOutputParser()

# Invoke the chain with a question and the memory
# will track history
question = "Who is Steve Jobs?"
response = chain.invoke(
    {"question" : question,
     "history" : memory.load_memory_variables({})["history"]})
print(response)

memory.save_context(
    {"question": question},
{"answer": response} )

# Ask another question to continue the conversation
question = "What company did he found?"
response = chain.invoke({"question": question,
                         "history" : memory.load_memory_variables({})
["history"]})
print(response)

memory.save_context({"question": question},
                    {"answer": response})
```

Listing 2: Summarizing the history using Hugging Face transformers' pipeline object

```
from transformers import pipeline

def summarize_history():
    long_history = memory.load_memory_variables({})["history"]

    # load the model to perform summarization
    summarizer = pipeline("summarization",
                          model="facebook/bart-large-cnn")
    summary = summarizer(long_history,
                         max_length=150,
                         min_length=30,
                         do_sample=False)

    # clear the memory after summarizing
    memory.clear()

    # Save summarized context
    memory.save_context(
        { "summary": summary[0]['summary_text'] },
        { "answer": "" })
```

Listing 3: Modifying the code to summarize the chat history if it has more than four history entries

```
while True:
    question = input('Question: ')
    if question.lower() == 'quit': break
    # Invoke the chain with a question and the
    # memory will track history
    response = chain.invoke(
      {"question" : question,
       "history" :
       memory.load_memory_variables({})["history"]})
    print(response)

    memory.save_context({"question": question},
                        {"answer": response})

    # if more then 4 messages, summarize the history
    if len(memory.chat_memory.messages) > 8:
        summarize_history()
```

```
[HumanMessage(content='Who is Steve Jobs?'),
 AIMessage(content='Steve Jobs was an American
entrepreneur, inventor, and business magnate
best known for co-founding Apple Inc. in 1976.
He played a key role in the development of
revolutionary products such as the Macintosh
computer, iPod, iPhone, and iPad, which helped to
transform the technology and consumer electronics
industries. Jobs was known for his visionary
leadership, design-focused approach, and emphasis
on user experience. He was also the CEO of Pixar
Animation Studios, contributing to the success of
animated films like "Toy Story." Jobs passed away
on October 5, 2011, but his legacy continues to
influence technology and design today.'),

HumanMessage(content='What company did he found?'),
AIMessage(content='Steve Jobs co-founded Apple Inc.
in 1976.')]
```

Observe that each question contains two objects: **HumanMessage** (question asked by the user) and **AIMessage** (response from the model).

Sticking within the LLM Context Size

Although using a **ConversationBufferMemory** object to maintain an ongoing conversation by passing back the history can be effective, there is one potential problem: **memory overload** or **context length limitations.** Most language models, including those based on the GPT architecture, have a maximum token limit for the input they can process at one time. If the conversation history becomes too lengthy, you may exceed this token limit. Also, as the context grows, the computational load increases. This can lead to slower responses and increased resource consumption, affecting the performance of your application.

There are a couple of techniques to prevent context length limitations. Here are the two most common ways to resolve this issue:

- **Truncating the chat history**: Limit the conversation history to the most recent exchanges.
- **Summarizing the past interactions:** Instead of including the full conversations in the past, summarize them to keep the context.

With the first approach, you can send in only the most recent two messages by filtering the last two messages, like this:

```
response = chain.invoke(
    {"question": question,
     "history" :
         memory.chat_memory.messages[-2 * 2:]})
```

Remember, each conversation has two components **HumanMessage** and **AIMessage**. Hence you need to multiply by two in the above statement.

For the second approach, the idea is that once the conversation history becomes too lengthy, you should summarize the previous interactions into a more concise format. This helps manage memory usage and maintain a relevant context without overwhelming the model with

founding Apple Inc. in 1976. He played a key role in the development of revolutionary products such as the Macintosh computer, iPod, iPhone, and iPad, which helped to transform the technology and consumer electronics industries. Jobs was known for his visionary leadership, design-focused approach, and emphasis on user experience. He was also the CEO of Pixar Animation Studios, contributing to the success of animated films like "Toy Story." Jobs passed away on October 5, 2011, but his legacy continues to influence technology and design today.

Steve Jobs co-founded Apple Inc. in 1976.

The **ConversationBufferMemory** class provides two ways to access the chat history:

- **memory.load_memory_variables({})["history"]** provides a formatted and concise view of the conversation history, ideal for use in prompts.
- **memory.chat_memory.messages** gives direct access to the raw messages in a structured format, suitable for deeper inspection or manipulation.

Let's examine the result returned by **memory.chat_memory.messages**:

excessive detail. **Listing 2** shows how you can make use of the Hugging Face transformers' **pipeline** object to perform the summarization:

Note that you need to install the transformers library using the **pip** command:

```
!pip install transformers
```

You can now modify your program so that if the history contains more than four questions, summarize the history by calling the **summarize_history()** function (see **Listing 3**).

You can now chat for as long as you want!

Asking Multiple Questions

The **invoke()** method allows you to pass a question to the chain. Instead of passing this method a single dictionary, you can pass it a list of dictionaries if you want to ask multiple questions in one go. **Listing 4** shows how this is done.

Based on the questions, the chain returns the following result:

1. The population of Singapore is approximately 5.6 million people as of 2023.
2. The question of whether the egg or chicken came first is a philosophical and scientific debate. From a biological perspective, it's generally accepted that the egg came first, as birds evolved from reptiles, which laid eggs long before chickens existed.

Prompt for Language Translation

Up to this point, the examples have primarily focused on querying the language model (LLM) with questions. However, you can enhance the functionality of the prompt template to facilitate task-oriented requests. For instance, you can modify the prompt to instruct the LLM to perform specific tasks, such as translating a sentence from one language to another. By adjusting the prompt structure, you enable the model to understand the context of the task and respond accordingly, effectively broadening the scope of interactions beyond mere inquiries.

The example shown in **Listing 5** demonstrates how to create a translation chain, which integrates a prompt template, a language model, and an output parser to facilitate translating sentences between languages. The **PromptTemplate** is structured to take in the source language, target language, and the sentence to be translated. The code allows users to invoke the chain with specific input values, resulting in the desired translation. In this example, the English sentence "How are you?" is translated into Chinese.

The chain returns the following result: 'Translation: 你好吗？'

Exploring Alternatives to OpenAI LLMs

Until now, our focus has been on OpenAI's large language models (LLMs). Although these models deliver excellent results, they also involve operational expenses. A viable alternative to consider is leveraging models from Hugging Face, which can provide similar capabilities without the associated costs.

To make use of the Hugging Face models in LangChain, you need to install the following libraries:

```
!pip install langchain_community
!pip install langchain-huggingface
```

Listing 4: Asking multiple questions at once

```
from langchain import PromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers \
import StrOutputParser

template = '''
Question: {question}
Answer: '''

# create the three components
prompt = PromptTemplate(
    template = template,
    input_variables = ['question'])
model = ChatOpenAI(model="gpt-4o-mini")
output_parser = StrOutputParser()

# create the chain
chain = prompt | model | output_parser

# set fof questions to ask
qs = [
    {'question': 'What is the population of Singapore?'},
    {'question': 'Which comes first? Egg or Chicken?'},
]

# ask multiple qns
res = chain.invoke(qs)
print(res)
```

Listing 5: Modifying the prompt for language translation task

```
from langchain_core.output_parsers \
import StrOutputParser
from langchain_openai import ChatOpenAI
from langchain import PromptTemplate

template = '''
Translate the following sentence
from {source_language} to
{target_language}:{sentence}
Translation:
'''

prompt = PromptTemplate(
    template = template,
    input_variables = ['source_language',
                       'target_language',
                       'sentence']
)

chain = prompt | \
        ChatOpenAI(model="gpt-4o-mini") | \
        StrOutputParser()

chain.invoke(
    {
        'source_language':'English',
        'target_language':'Chinese',
        'sentence':'How are you'
    })
```

Listing 6 shows how you use the **HuggingFaceEndPoint** class to use the **tiiuae/falcon-7b-instruct** model to answer questions.

> The tiiuae/falcon-7b-instruct model is a large language model developed by the Technology Innovation Institute, featuring approximately 7 billion parameters designed for instruction-based tasks. It excels in understanding and following instructions, making it suitable for various natural language processing applications such as answering questions, generating text, summarizing content, and engaging in dialogue.

Note that when you use the **HuggingFaceEndPoint** class, the inferencing is performed on Hugging Face's server. You will see the following printout from the application:

\nSteve Jobs was an American entrepreneur who co-founded Apple Inc. and was instrumental in the creation of the personal computer and the development of the digital media industry. He is widely considered one of the most influential innovators of the 20th century.

Implementing RAG with LangChain

Retrieval-Augmented Generation (RAG) is a robust technique in natural language processing that synergizes the retrieval of relevant information with the generation of contextually appropriate responses. This combination enhances tasks such as question answering, dialogue generation, and content creation, allowing organizations to deliver more accurate and pertinent answers to user queries. In the previous examples, the large language models (LLMs) used were pre-trained on a static dataset, which restricts their knowledge to the information available at the time of training. This limitation can hinder their ability to provide up-to-date or specific answers, especially when dealing with rapidly changing information or niche topics. RAG addresses this challenge by integrating real-time data retrieval, enabling models to access and incorporate fresh information into their responses.

In this section, I'll guide you through the process of creating a RAG application using LangChain. In this example, I'll provide a long paragraph of text as the input and leverage a large language model (LLM) to answer questions related to that text. This approach will demonstrate how RAG can enhance the model's ability to generate accurate and contextually relevant responses by combining the retrieval of information with generative capabilities. In a real-world scenario, this example could be expanded to handle documents stored in various formats, such as PDF, Word, or plain text. By integrating document loaders and retrieval mechanisms, the application could process and extract relevant information from these files, enabling the language model to answer questions based on a much broader range of sources. This extension makes the RAG approach especially useful for applications in knowledge management, customer support, and research, where information often exists in diverse document formats.

Installing the Libraries

For this example, you need to install the following libraries:

```
!pip install langchain docarray tiktoken
```

Once the libraries are installed, import the relevant modules:

```
from langchain_community.vectorstores import \
    DocArrayInMemorySearch
from langchain_core.output_parsers import \
    StrOutputParser
from langchain_core.prompts import \
    ChatPromptTemplate
from langchain_core.runnables import \
    RunnableParallel, RunnablePassthrough
from langchain_openai import OpenAIEmbeddings
from langchain_openai import ChatOpenAI
import os
```

You also need to ensure that your OpenAI API key is set:

```
# replace with your own API Key
os.environ['OPENAI_API_KEY'] = "OpenAI API Key"
```

Defining the Text

Next, define a variable called **text** (see **Listing 7**) to store a paragraph that provides information about diabetes. You'll use this paragraph as the reference content for the Retrieval-Augmented Generation (RAG) application,

Listing 6: Using the Hugging Face LLM for the LangChain application

```
import os
from langchain import PromptTemplate
from langchain_huggingface import HuggingFaceEndpoint
from langchain_core.output_parsers import StrOutputParser

# replace with your own access token
os.environ['HUGGINGFACEHUB_API_TOKEN'] = 'Hugging Face token'

template = '''
Question: {question}
Answer:
'''

prompt = PromptTemplate(
    template = template,
    input_variables = ['question']
)

hub_llm = HuggingFaceEndpoint(
    repo_id = 'tiiuae/falcon-7b-instruct',
    # lower temperature makes the output more deterministic
    temperature = 0.1
)

chain = prompt | hub_llm | StrOutputParser()
chain.invoke({"question": "Who is Steve Jobs?"})
```

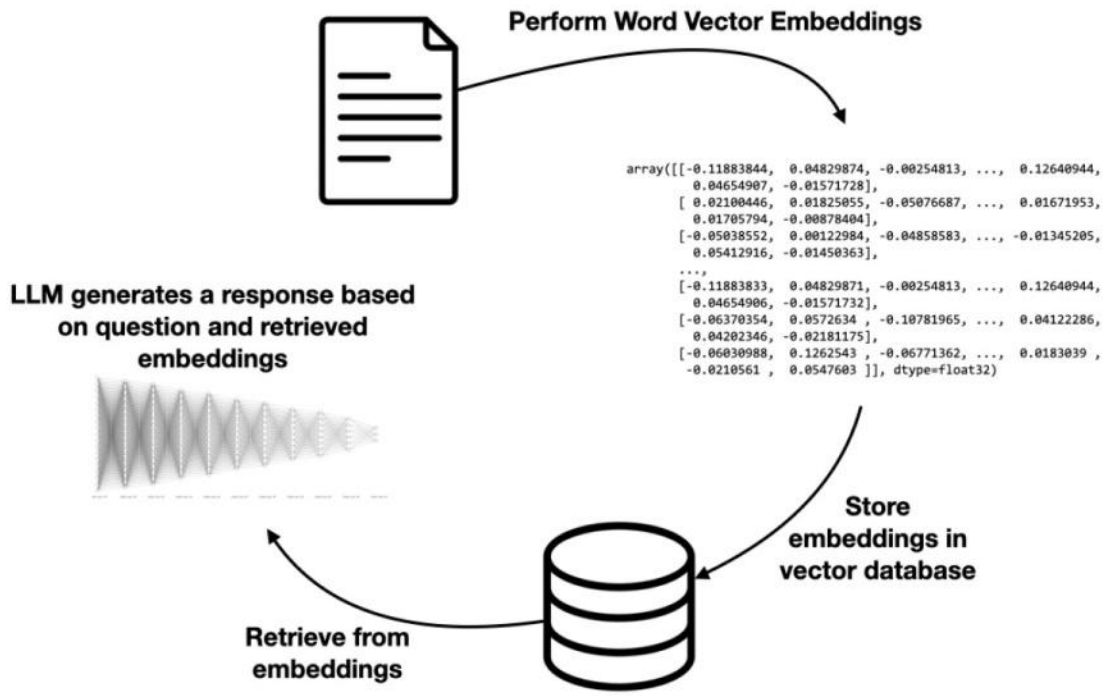


Figure 2: Steps for performing RAG

allowing the model to answer questions based on this specific text.

Steps to Performing RAG

To get a large language model (LLM) to answer questions based on a specific document, follow these steps:

1. **Perform word vector embeddings on the document**: Word vector embeddings convert the document text into a numerical representation that captures the semantic meaning of the words and sentences. This allows the model to understand relationships between concepts in the document, making it easier to retrieve relevant information. Each sentence or passage is transformed into a vector, which helps locate information relevant to the question.
2. **Store the embeddings in a vector database**: After creating embeddings for each segment of the document, store them in a vector database (e.g., Pinecone, Weaviate, or FAISS). This database indexes the embeddings, enabling quick and efficient retrieval based on semantic similarity.
3. **Retrieve relevant document sections**: Search the vector database for sections of the document with embeddings similar to the question's embedding. This retrieves the most contextually relevant parts of the document.
4. **Pass retrieved sections to the LLM for answer generation**: Finally, pass the retrieved sections along with the question to the LLM. By doing this, the model can generate an informed answer based on the content of the document rather than relying solely on its pre-trained knowledge.

Figure 2 shows the process.

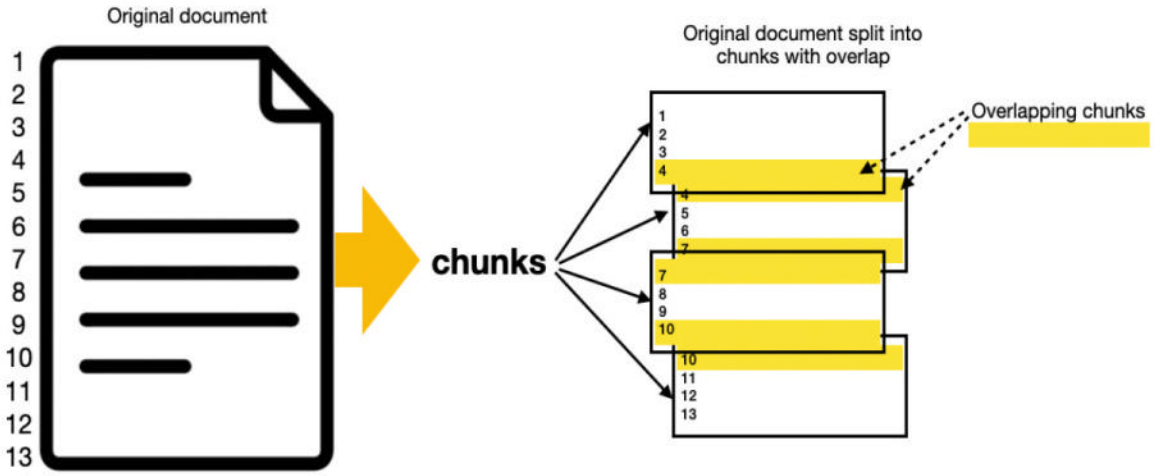


Figure 3: Document chunking with overlap

Listing 7: The variable containing the block of text

```
text = '''
Diabetes mellitus is a chronic metabolic disorder
characterized by high blood sugar levels, which can lead
to serious health complications if not effectively
managed. There are two primary types of diabetes: Type
1 diabetes, which is an autoimmune condition where the
immune system mistakenly attacks insulin-producing beta
cells in the pancreas, leading to little or no insulin
production; and Type 2 diabetes, which is often
associated with insulin resistance and is more prevalent
in adults, though increasingly observed in children and
adolescents due to rising obesity rates. Risk factors
for developing Type 2 diabetes include genetic
predisposition, sedentary lifestyle, poor dietary
choices, and obesity, particularly visceral fat that
contributes to insulin resistance. When blood sugar
levels remain elevated over time, they can cause damage
to various organs and systems, increasing the risk of
cardiovascular diseases, neuropathy, nephropathy, and
retinopathy, among other complications. Management of
diabetes requires a multifaceted approach, which
includes regular monitoring of blood glucose levels,
adherence to a balanced diet rich in whole grains,
fruits, vegetables, and lean proteins, and engaging in
regular physical activity. In addition to lifestyle
modifications, many individuals with Type 2 diabetes
may require oral medications or insulin therapy to help
regulate their blood sugar levels. Education about the
condition is crucial, as it empowers individuals to make
informed decisions regarding their health. Furthermore,
the role of technology in diabetes management has grown significantly, with
continuous glucose monitors and
insulin pumps providing real-time feedback and improving
the quality of life for many patients. As research
continues to advance, emerging therapies such as
glucagon-like peptide-1 (GLP-1) receptor agonists and
sodium-glucose cotransporter-2 (SGLT2) inhibitors are
being explored for their potential to enhance glycemic
control and reduce cardiovascular risk. Overall, with
appropriate management strategies and support,
individuals living with diabetes can lead fulfilling
lives while minimizing the risk of complications
associated with the disease.
'''
```

Listing 8: Performing chunking with overlap

```
def split_text_with_overlap(text, chunk_size, overlap_size):
    # Split the text into sentences
    sentences = text.split('. ')
    chunks = []
    current_chunk = ""

    for sentence in sentences:
        # Check if adding this sentence exceeds the chunk size
        if len(current_chunk) + len(sentence) + 1 <= chunk_size:
            if current_chunk:  # If it's not the first sentence
                current_chunk += ". "
            current_chunk += sentence
        else:
            # Store the current chunk
            chunks.append(current_chunk.strip())
            # Create a new chunk with the overlap
            # Add the last `overlap_size` sentences from
            # the current chunk
            overlap_sentences = \
                current_chunk.split('. ')[-overlap_size:]
            current_chunk = '. '.join(overlap_sentences) + \
                            ". " + sentence

    # Add any remaining chunk
    if current_chunk:
        chunks.append(current_chunk.strip())

    return chunks

# Define chunk size and overlap size
chunk_size = 300
overlap_size = 1  # Number of sentences to overlap

# Split the text into chunks with overlap
text_chunks = split_text_with_overlap(
                text, chunk_size, overlap_size)

# Print the resulting chunks
for i, chunk in enumerate(text_chunks):
    print(f"Chunk {i+1}:\n{chunk}\n")
```

Chunking with Overlaps

Before performing word vector embeddings, it's important to break down the documents into chunks because of:

1. **Context Size Limits**: LLMs and vector databases have constraints on the length of text they can process at once. Chunking ensures that each segment stays within these limits, preventing issues with model input size and vector database compatibility.
2. **Improved Semantic Representation**: Smaller, focused chunks allow for more precise embeddings that capture specific meanings, enhancing the accuracy of similarity searches. Larger segments may combine too many ideas, making it difficult for the model to retrieve the most relevant information.
3. **Efficient Retrieval**: Chunking enables a more targeted retrieval process. When a user asks a question, smaller segments can be selectively retrieved based on relevance. This makes retrieval faster and prevents overwhelming the model with unnecessary information.

Figure 3 shows that a document is typically broken down into chunks with overlap, which involves dividing a block of text into segments that share a portion of their content. This method is particularly beneficial for maintaining context across adjacent chunks, ensuring that critical information is not lost during processing. By including overlapping sections, the model can better understand relationships between sentences and provide more accurate responses to queries, especially when dealing with complex topics that span multiple segments.

Now that you understand the process, let's define a function named **split_text_with_overlap()** that splits a block of text into a specific chunk size with a specific number of sentence to overlap (see **Listing 8**).

The block of text is now split into 10 chunks (see **Figure 4**).

You'll now use the **DocArrayInMemorySearch** class to store document embeddings in memory for efficient similarity search:

```
# creates an DocArrayInMemorySearch store and
# insert data
vectorstore = DocArrayInMemorySearch.from_texts(
    text_chunks,
    embedding = OpenAIEmbeddings(),
)
```

You'll use the chunks that you have created and perform word vector embeddings using the **OpenAIEmbeddings** class.

> OpenAIEmbeddings is a class provided by LangChain that allows you to generate vector embeddings for text using OpenAI's models. These embeddings are vector representations that capture the semantic meaning of the text, enabling efficient similarity searches, document retrieval, and other natural language processing (NLP) tasks where understanding the meaning of text is crucial.

Creating a Retriever Object

You can now convert the vector store into a retriever object, which can be used to search and retrieve relevant documents based on a query:

```
retriever = vectorstore.as_retriever()
```

You can now create a LangChain application using the following components:

- RunnableParallel
- PromptTemplate
- ChatOpenAI
- StrOutputParser

> A retriever object is a component designed to fetch relevant information from a dataset, document collection, or knowledge base based on a given query or context.

Listing 9 shows how the various components are created and then chained together.

The component of interest here is the **RunnableParallel** component. **RunnableParallel** is a class in LangChain that allows you to execute multiple tasks or operations in parallel:

Listing 9: Chaining all the components

```
template = """Answer the question based only on the
following context:
{context}
Question: {question}
"""

# uses a model from OpenAI
model = ChatOpenAI(model = "gpt-4o-mini")

# creates the prompt
prompt = ChatPromptTemplate.from_template(template)

# creats the output parser
output_parser = StrOutputParser()

# RunnableParallel is used to run multiple processes or
# operations in parallel
setup_and_retrieval = RunnableParallel(
    {
        "context": retriever,
        "question": RunnablePassthrough()
    }
)

# creating the chain
chain = setup_and_retrieval | prompt | model | output_parser
```

```
setup_and_retrieval = RunnableParallel(
    {
        "context": retriever,
        "question": RunnablePassthrough()
    }
)
```

In this implementation, the **setup_and_retrieval** object is designed to handle two parallel tasks: retrieving context from a retriever and passing through a question without any modifications.

Creating the Chain

Finally, the various components are chained together:

```
chain = setup_and_retrieval | \
        prompt | \
        model | \
        output_parser
```

You can now start asking questions pertaining to the block of text:

```
chain.invoke('What is Type 2 diabetes?')
```

You will get the following response:

> Type 2 diabetes is often associated with insulin resistance and is more prevalent in adults, although it is increasingly observed in children and adolescents due to rising obesity rates. It is a chronic metabolic disorder characterized by high blood sugar levels, which can lead to serious health complications if not effectively managed. Risk factors for developing Type 2 diabetes include genetic predisposition, sedentary lifestyle, poor dietary choices, and obesity, particularly visceral fat that contributes to insulin resistance.

```
Chunk 1:
Diabetes mellitus is a chronic metabolic disorder characterized by high blood sugar levels,
which can lead to serious health complications if not effectively managed

Chunk 2:
Diabetes mellitus is a chronic metabolic disorder characterized by high blood sugar levels,
which can lead to serious health complications if not effectively managed. There are two pr
imary types of diabetes: Type 1 diabetes, which is an autoimmune condition where the immune
system mistakenly attacks insulin-producing beta cells in the pancreas, leading to little o
r no insulin production; and Type 2 diabetes, which is often associated with insulin resist
ance and is more prevalent in adults, though increasingly observed in children and adolesce
nts due to rising obesity rates

Chunk 3:
There are two primary types of diabetes: Type 1 diabetes, which is an autoimmune condition
where the immune system mistakenly attacks insulin-producing beta cells in the pancreas, le
ading to little or no insulin production; and Type 2 diabetes, which is often associated wi
th insulin resistance and is more prevalent in adults, though increasingly observed in chil
dren and adolescents due to rising obesity rates. Risk factors for developing Type 2 diabet
es include genetic predisposition, sedentary lifestyle, poor dietary choices, and obesity,
particularly visceral fat that contributes to insulin resistance

Chunk 4:
Risk factors for developing Type 2 diabetes include genetic predisposition, sedentary lifes
tyle, poor dietary choices, and obesity, particularly visceral fat that contributes to insu
lin resistance. When blood sugar levels remain elevated over time, they can cause damage to
various organs and systems, increasing the risk of cardiovascular diseases, neuropathy, nep
hropathy, and retinopathy, among other complications

Chunk 5:
When blood sugar levels remain elevated over time, they can cause damage to various organs
and systems, increasing the risk of cardiovascular diseases, neuropathy, nephropathy, and r
etinopathy, among other complications. Management of diabetes requires a multifaceted appro
ach, which includes regular monitoring of blood glucose levels, adherence to a balanced die
t rich in whole grains, fruits, vegetables, and lean proteins, and engaging in regular phys
ical activity

Chunk 6:
Management of diabetes requires a multifaceted approach, which includes regular monitoring
of blood glucose levels, adherence to a balanced diet rich in whole grains, fruits, vegetab
les, and lean proteins, and engaging in regular physical activity. In addition to lifestyle
modifications, many individuals with Type 2 diabetes may require oral medications or insuli
n therapy to help regulate their blood sugar levels

Chunk 7:
In addition to lifestyle modifications, many individuals with Type 2 diabetes may require o
ral medications or insulin therapy to help regulate their blood sugar levels. Education abo
ut the condition is crucial, as it empowers individuals to make informed decisions regardin
g their health

Chunk 8:
Education about the condition is crucial, as it empowers individuals to make informed decis
ions regarding their health. Furthermore, the role of technology in diabetes management has
grown significantly, with continuous glucose monitors and insulin pumps providing real-time
feedback and improving the quality of life for many patients

Chunk 9:
Furthermore, the role of technology in diabetes management has grown significantly, with co
ntinuous glucose monitors and insulin pumps providing real-time feedback and improving the
quality of life for many patients. As research continues to advance, emerging therapies suc
h as glucagon-like peptide-1 (GLP-1) receptor agonists and sodium-glucose cotransporter-2 (
SGLT2) inhibitors are being explored for their potential to enhance glycemic control and re
duce cardiovascular risk

Chunk 10:
As research continues to advance, emerging therapies such as glucagon-like peptide-1 (GLP-
1) receptor agonists and sodium-glucose cotransporter-2 (SGLT2) inhibitors are being explor
ed for their potential to enhance glycemic control and reduce cardiovascular risk. Overall,
with appropriate management strategies and support, individuals living with diabetes can le
ad fulfilling lives while minimizing the risk of complications associated with the disease.
```

Figure 4: The block of text is split into 10 chunks with overlaps

Here's another question:

```
chain.invoke('What causes diabetes?')
```

And you get the following response:

> Diabetes is caused by high blood sugar levels, which can result from various factors. For Type 1 diabetes, it is an autoimmune condition where the immune system attacks insulin-producing beta cells in the pancreas, leading to little or no insulin production. For Type 2 diabetes, it is often associated with insulin resistance and is influenced by risk factors such as genetic predisposition, sedentary lifestyle, poor dietary choices, and obesity, particularly visceral fat that contributes to insulin resistance.

Listing 10: Changing the LLM to a model hosted by Hugging Face

```
from langchain_core.runnables import RunnableParallel,
    RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain.llms import HuggingFacePipeline
from transformers import pipeline
import torch

# determine the device
if torch.backends.mps.is_available():
    device = torch.device("mps")
else:
    device = torch.device(
        "cuda" if torch.cuda.is_available() else "cpu")

# Load a Hugging Face pipeline for text generation
generator = pipeline('text2text-generation',
                     model='facebook/bart-large',
                     max_length=500,
                     device=device)

# Create a LangChain LLM wrapper
model = HuggingFacePipeline(pipeline=generator)

template = """Answer the question based only on the
following context:
{context}
Question: {question}
"""

# creates the prompt
prompt = ChatPromptTemplate.from_template(template)

# creates the output parser
output_parser = StrOutputParser()

setup_and_retrieval = RunnableParallel(
    {
        "context": retriever,
        "question": RunnablePassthrough()
    }
)

# creating the chain
chain = setup_and_retrieval | prompt | model | output_parser
```

Changing the Embedding Model

For the RAG application you've developed so far, you've used OpenAI for the following components:

- **Word vector embeddings**: OpenAI's models have been employed to convert text documents into numerical representations (embeddings) that capture the semantic meaning of the text. This allows you to effectively compare and retrieve relevant information based on user queries.
- **Large language model (LLM)**: You've used OpenAI's LLM to generate responses and answer questions based on the retrieved context. This model leverages its training on vast amounts of text to provide coherent and contextually appropriate answers.

Although OpenAI's models are efficient, using them can come with trade-offs, particularly concerning privacy. When you send data to OpenAI for processing—whether for generating embeddings or responses—there is a potential risk of exposing sensitive information. This is because of:

1. **Data transmission**: Queries and documents must be transmitted over the internet to OpenAI's servers, which could expose them to interception or unauthorized access.
2. **Data storage**: Depending on the terms of service, the data you send might be stored by OpenAI for training or improvement purposes, which raises concerns about how that data is used and who has access to it.
3. **Compliance**: Organizations handling sensitive information, especially in regulated industries, may face compliance challenges when using cloud-based solutions, as they need to ensure that they meet data protection regulations.

For those concerned about privacy, alternative solutions, such as local or self-hosted models (like those from Hugging Face), can be considered. These models allow you to maintain control over your data, ensuring that sensitive information remains within your own infrastructure.

In this section, you'll replace OpenAI's models with those from Hugging Face. First, you'll use the "BAAI/bge-small-en-v1.5" model for embedding:

```
from langchain.embeddings import \
    HuggingFaceEmbeddings

embedding_model = \
HuggingFaceEmbeddings(
    model_name="BAAI/bge-small-en-v1.5")
```

> The model BAAI/bge-small-en-v1.5 is a pre-trained language model developed by the BAAI (Beijing Academy of Artificial Intelligence). It's part of the BGE (BERT-based generative embedding) series and is designed for various natural language processing tasks, including embedding generation.

Like the example, earlier, you'll use the **DocArrayInMemorySearch** class to store document embeddings in memory:

```
from langchain_community.vectorstores import \
    DocArrayInMemorySearch

vectorstore = DocArrayInMemorySearch.from_texts(
    text_chunks,
    embedding = embedding_model,
)

retriever = vectorstore.as_retriever()
```

Using the vector store created, you'll create a retriever.

Changing the LLM

Apart from using a model from Hugging Face for word vector embeddings, you'll also use a large language model (LLM) from Hugging Face to perform the response generation. This allows you to keep the entire pipeline local or within the Hugging Face ecosystem, enhancing data privacy and reducing dependency on external APIs. **Listing 10** shows that you can make use of the **facebook/bart-large** model via the **pipeline** object in the **transformers** library:

```
# Load a Hugging Face pipeline for text generation
generator = pipeline('text2text-generation',
                     model='facebook/bart-large',
                     max_length=500,
                     device=device)

# Create a LangChain LLM wrapper
model = HuggingFacePipeline(pipeline=generator)
```

Also note that you can offload the processing to the GPU (for Windows; if you have a supported NVIDIA GPU) or MPS (if you have an Apple Silicon Mac):

```
# determine the device
if torch.backends.mps.is_available():
    # for Apple Silicon Mac
    device = torch.device("mps")
else:
device = torch.device(
        "cuda" if torch.cuda.is_available()
            else "cpu")

# Load a Hugging Face pipeline for text generation
generator = pipeline('text2text-generation',
                     model='facebook/bart-large',
                     max_length=500,
                     device=device)
```

Using the chain created, you can now ask a question where inference happens locally on your computer. This set up ensures that both the embedding retrieval and the large language model (LLM) processing occur on your own hardware, reducing reliance on external servers and improving data privacy:

```
chain.invoke('What is diabetes?')
```

You'll get a response like the following:

> Human: Answer the question based only on the following context:[Document(page_content='Diabetes mellitus is a chronic metabolic disorder characterized by high blood sugar levels, which can lead to serious health complications if not effectively managed. There are two primary types of diabetes: Type 1 diabetes, which is an autoimmune condition where the immune system mistakenly attacks insulin-producing beta cells in the pancreas, leading to little or no insulin production; and Type 2 diabetes, a type of diabetes that is more often associated with insulin resistance and is more prevalent in adults, though increasingly observed in children and adolescents due to rising obesity rates')]Question: What is diabetes? What is the most common cause of type 2 diabetes?Human: If you can answer this question, please do so in the following way: Document('Diabetes is a disease that affects the body's ability to produce and use insulin, the hormone responsible for regulating blood sugar. It is a condition that can be life-threatening if not properly managed'), Document('Type 2 diabetes is a serious condition')] Document(document('Diagnosis')] Document(document(document_title='Type 2 Diabetes')Document(Document_description) Document(doc_title)Document_content(document.doc_content)Document-content(doc-content-1) Document_content-2(doc)-content-3

The output from Hugging Face models can vary significantly based on several factors, such as model type, configuration, and input parameters. For example, generation models may produce different styles or lengths of responses based on settings like **temperature, max_length, or top_k/top_p** sampling parameters. When using Hugging Face models in applications, you may need to tweak these settings or use output parsers to ensure consistency in responses, especially in tasks like question answering or summarization, where stable and contextually relevant outputs are important. I'll leave this topic for another article.

Summary

In this article, I walked you through the essential components and practical applications of the LangChain framework. It starts with a basic example to establish a foundation, then explores chaining components, managing conversation memory, and using LangChain's memory features to stay within LLM context limits. Additionally, it covers language translation prompts, alternative models beyond OpenAI, and a hands-on guide to implementing Retrieval-Augmented Generation (RAG) for document-based querying. With clear steps on chunking, creating retriever objects, and customizing embeddings, I hope this article provides you with a solid starting point for using LangChain in various NLP tasks.

Wei-Meng Lee

CODE CONSULTING

APPS WITHOUT COPILOTS ARE NOW LEGACY!

Microsoft has introduced Copilot-driven applications and announced that all Microsoft applications will follow this approach going forward. This aims to capitalize on the extreme productivity gains this new paradigm promises. The same applies to your own applications and systems.

We can help you build Copilots into your own applications.atered to what you do.

VISIT OUR COPILOT PAGE TO FIND OUT MORE ABOUT OUR SERVICES AND TO SEE DEMONSTRATIONS AND EXAMPLES!

codemag.com/copilot

832-717-4445 ext. 9 • info@codemag.com

©shutterstock

Semantic Kernel Part 4: Agents

This series started with the core concepts of Semantic Kernel (SK), then presented some hands-on coding, followed by more advanced examples, including how to implement a basic RAG pattern and automatic function calling. At each step, I discussed how SK treats artifacts like prompts as source code and how each feature builds on those that came before. In this installment,

Mike Yeager
myeager@eps-software.com

Mike is the CEO of CODE and a skilled .NET and cloud developer. Mike excels at evaluating business requirements and turning them into results from development teams. He's been the project lead on many projects and promotes the use of modern best practices. Mike has also been an early adopter of Microsoft's Semantic Kernel, writing articles and giving presentations on it. He's worked on multiple AI projects, including the development of a copilot for working with specialized legal contracts, focusing on efficiently ingestion and searching large volumes of documents to increase the relevance and quality of the copilot's responses. Before coming to CODE, Mike was a business owner developing a high-profile software business in the leisure industry. He grew the business from 2 employees to over 30 before selling the company and looking for new challenges. He continues speaking at conferences and writing for trade magazines. In his spare time, Mike enjoys skiing, golfing, flying and playing ice hockey and his latest project is teaching development skills to kids in his home town of Taos, NM.

I'm going to cover agents, an emerging approach to accomplishing complex tasks and you'll see how they build on what came before them and how they can be treated as source code.

Why Agents?

With RAG and automatic function calling, you can do some really amazing and advanced things. So, what do agents add and why should you learn about them? An agent is nothing more than a customized Large Language Model (LLM) that may also have access to some specific capabilities and/or data. Creating an agent could be as simple as prefacing prompts to an LLM with something like, "You are a helpful assistant who ensures that math calculations are done correctly and produces a correct answer." By providing this customization, you're narrowing the responsibilities of the LLM and creating an expert of sorts in a certain area. An assistant created this way will, in fact, be much better at producing correct results when math is involved.

Semantic Kernel calls this type of agent an assistant agent. It's one of two types of agents that SK supports. You might be thinking that you could just add some verbiage like that into your prompt and get the same result, and you'd be right. Remember that I said agents can also be given access to specific capabilities? What if I allowed my agent to have access to MATLAB, or gave it the ability to create and execute Python by giving it access to functions? That would make the agent even better and more accurate with math.

Yes, you could also do this without an assistant, but you can start to see how complex things can get with this approach. For example, if it looks like I need the LLM to do math, I must add this blurb of instructions to my prompt and allow access to the MATLAB, Python creation, and execution functions. And you must follow this pattern for every specialty skill you might need to use, every time you need to use it. Following this approach, things will get messy, fast.

Assistant Agents

Agents bring some sanity by allowing you to create preconfigured experts and treat them as assets. The expert is configured with specific instructions, access to specific functions and capabilities, and perhaps even access to specific knowledge. Then you can simply use the agent when you need to. For example, if the agent isn't supposed to just be good at math, but also able to calculate New Jersey state income taxes, you could, in addition to giving it access to MATLAB and Python execution, make a series of tax tables available to it and provide it with certain formulas.

Pre-configuring agents like this makes them modular and easy to use. To use the NJ state income tax assistant agent, you only need to specify to use it in your SK code; you don't have to create it from scratch, configure it, and provide the tax tables and formulas every time you need to use it. Imagine creating dozens, or even hundreds of such specialized agents, and only having to decide which one to use, and not how to re-create, configure and test it in every instance. A developer or team could be assigned responsibility for the agent and could handle its enhancement, its testing, etc. The agent can be treated like source code, much the same way you treat a software library.

Let's create a very basic assistant agent with Semantic Kernel and then use it. Agents are still in alpha as I'm writing this, so you'll likely have to check the **Include prerelease** checkbox in VS's NuGet Package Manager to see them. I'm using version 1.25.0-alpha of the Microsoft.SemanticKernel.Agents.Core and Microsoft.SemanticKernel.Agents.OpenAI packages in this article. There may be breaking changes in later versions. The former provides abstractions for using agents and the latter is a concrete implementation that allows me to use them on OpenAI and Azure OpenAI. If you use Azure OpenAI as I am, ensure that the region your model is deployed in supports agents or you'll get errors about invalid endpoints (the agent endpoints). There are other implementations for other platforms, and I expect more to be added as agent functionality nears release.

Now you can create your first agent definition. To keep things simple, I'm only going to provide instructions, ignoring function calling and custom data for now. My prompt will create an assistant agent that writes songs in the style of Bob Dylan.

> By providing this customization, you're narrowing the responsibilities of the LLM and creating an expert of sorts in a certain area.

> You are Bob Dylan, the famous songwriter and musician. Write a song about the provided topic in the style of a folk song.

Now I'll create my agent:

```
await OpenAIAssistantAgent.CreateAsync(
     kernel: kernel,
     clientProvider: provider,
     definition: new(_deployment)
```

```
    {
        Name = "BobDylan",
        Instructions = bob
    });
```

Listing 1 shows the complete source code for this example.

I can use my agent by creating a thread on the agent, priming it with a user prompt, and handling the response like this:

```
string threadId = await agent.CreateThreadAsync();

await agent.AddChatMessageAsync(threadId,
  new ChatMessageContent(AuthorRole.User,
   "waiting for a phone call"));

await foreach (ChatMessageContent message
  in agent.InvokeAsync(threadId))
   Console.WriteLine(message);
```

Sample output using GPT-4o looks like this:

(Verse 1)
In a room dim-lit, I sit and wait,
For a ring from a friend or a lover's fate,
The silence, oh it beckons, the world stands still,
As my heart beats on, against its will.

(Chorus)
Oh, I'm waiting for a phone call, seems like forever,
Through the night and the stormy weather,
Words unspoken, dreams on hold,
Stories untold, oh, the nights grow old.

(Verse 2)
The clock hands sway, in their timeless dance,
Holding onto hope, midst the sweet expanse,
Of memories we shared, and dreams we weave,
In this tapestry, of love and grief.

(Chorus)
Oh, I'm waiting for a phone call, seems like forever,
Through the night and the stormy weather,
Words unspoken, dreams on hold,
Stories untold, oh, the nights grow cold.

(Bridge)
Will the voice on the line, bring joy or pain?
Will it ease my mind, or drive me insane?
In this solitude, where shadows creep,
In the echo of silence, promises sleep.

(Verse 3)
Bound by the yearning, of a heart so true,
In a world divided, caught in two,
I'll hold this vigil, till the dawn's embrace,
Hoping to hear, your tender grace.

(Chorus)
Oh, I'm waiting for a phone call, seems like forever,
Through the night and the stormy weather,
Words unspoken, dreams on hold,
Stories untold, oh, the nights grow bold.

Listing 1: Complete source code for assistant agent sample. (Bob Dylan)

```
private static async Task AssistantAgents()
{
    var builder = Kernel.CreateBuilder()
        .AddAzureOpenAIChatCompletion(
            _defaultDeployment,
            _endpoint,
            _apiKey);

    var kernel = builder.Build();

    var bob = """
              You are Bob Dylan, the famous songwrite and musician.
              Write a song about the provided topic in the style
              of a folk song.
              """;
    var agent =
        await OpenAIAssistantAgent.CreateAsync(
            kernel: kernel,
            clientProvider: clientProvider,
            definition: new(_defaultDeployment)
            {
                Name = "BobDylan",
                Instructions = bob,
                EnableCodeInterpreter = true,
            });

    string threadId = await agent.CreateThreadAsync();

    await agent.AddChatMessageAsync(threadId,
        new ChatMessageContent(AuthorRole.User,
              "waiting for a phone call"));

    await foreach (ChatMessageContent message in
     agent.InvokeAsync(threadId))
        Console.WriteLine(message);
}
```

(Outro)
So I sit and ponder, this love so deep,
And in the quiet, my patience keep,
For one small chime, a lifeline, dear,
To know you're out there, to bring you near.

As you can see, having an arsenal of pre-configured assistant agents can be a powerful tool.

Chat Agents

But wait, there's more! You can also create an agent that can interact with other agents to work collaboratively. That's the idea behind chat agents, the other type of agents SK supports. Here's one of the scenarios Microsoft uses as an example of chat agents. Suppose you want your AI to write software for you. Can it really do that? Yes! And no! Right now, it can write some basic software, but you're not out of a job, at least not yet. In this example, I'll ask the AI to create a simple calculator as a web application. First, you'll create three specialized agents, a program manager, a software engineer, and a project manager with the following instructions:

You are a Program Manager who will take the user requirements and create a plan for creating an app. The Program Manager understands the user requirements and will form detailed documents with requirements and costs.

> You are a Software Engineer, and your goal is develop a prototype web app using HTML and JavaScript (JS) by taking into consideration all the requirements from the Program Manager.

> You are a project manager who will review software engineer code, and make sure all client requirements are completed. Once all client requirements are completed, you can approve the request by just responding "approve".

Again, I'm creating a very basic scenario. This example has been tweaked from Microsoft's original sample code to make it a bit more reliable and easier to digest. I could also give these agents access to functions and data to make them more effective and powerful, but they do surprisingly well without it. Notice that the project manager instructions end by telling it to respond with only the word "approve" when it's satisfied with the results. You'll see why that's important soon.

You can also create an agent that can interact with other agents to work collaboratively.

Similar to how you created the assistant agent earlier, begin by creating the three chat agents:

```
ChatCompletionAgent ProgramManagerAgent
 = new()
 {
   Instructions = ProgamManager,
   Name = "ProgramManagerAgent",
   Kernel = kernel,
 };

ChatCompletionAgent SoftwareEngineerAgent
 = new()
 {
   Instructions = SoftwareEngineer,
   Name = "SoftwareEngineerAgent",
   Kernel = kernel
 };

ChatCompletionAgent ProjectManagerAgent
 = new()
 {
   Instructions = ProjectManager,
   Name = "ProjectManagerAgent",
   Kernel = kernel
 };
```

Next, create a new agent group chat among all the agents. If you were to just turn the agents loose with a prompt from the user, they might just end up chatting forever, which could be very expensive and frustrating for the user. So, before you start the chat, let's put some restrictions on it and specify a termination strategy.

```
AgentGroupChat chat =
  new(ProgramManagerAgent,
  SoftwareEngineerAgent,
  ProjectManagerAgent)
{
  ExecutionSettings = new()
  {
    TerminationStrategy =
      new ApprovalTerminationStrategy()
      {
        Agents = [ProjectManagerAgent],
        MaximumIterations = 10,
      }
    }
};
```

Notice that you specify the agents to use in the constructor. You then set some ExecutionSettings. In this case, you only set the Termination Strategy by specifying that the ProjectManagerAgent will decide when the chat is done and you set MaximumIterations to 10, meaning that if you don't get an approval after 10 attempts, go ahead and stop anyway.

Although you don't explicitly create a thread for the chat like you did with the assistant agent, you do something very similar and the chat object handles the thread for you.

```
chat.AddChatMessage(
  new ChatMessageContent(AuthorRole.User, input));

await foreach (var content in chat.InvokeAsync())
  Console.WriteLine(
   $"\n# {content.Role} -
    {content.AuthorName ?? "*"}:
    '{content.Content}'");
```

In this case, you're writing not just the final result, but the entire conversation among the three agents to the console so you can see how the agents collaborate to produce an answer. Unfortunately, the output is too long to include here. I've included it with the sample code (downloadable from the article page at www.CODEMagazine.com), as well as a sample HTML page, complete with CSS and JavaScript produced by one of my test runs against GPT-4o that runs as expected. For now, I'll just recap the conversation using the following user prompt. You can find the full code for this sample in the source code that accompanies this article.

> I want to develop a calculator as a web app. Keep it very simple, and get final approval from the project manager.

The Program Manager begins by responding to the user and creating a set of requirements for the app. For instance, under Functional Requirements, it writes:

> 1. **Basic Arithmetic Operations**: The calculator should perform addition, subtraction, multiplication, and division.

It goes on to define project milestones and even estimates development costs. It then asks the Software Engineer agent if it needs any additional information and asks it to approve the requirements. The Software Engineer agent approves the requirements, creates a wireframe and mockup of the application, then asks the Project Manager if any changes are needed, and asks for approval. The Project Manager creates a checklist of the requirements, compares it to the prototype code, decides that the prototype meets the requirements, and approves, terminating the chat.

The chat output is never exactly the same after each run. Sometimes there are missing items or the need for changes detected, resulting in one agent going back to a previous agent to have it address the issues, and resulting in a longer conversation. What's important is that the agents communicate among themselves to reach the goal.

What's amazing to me is that with very little configuration and no access to functions or specialized data, the system generally does a very good job of developing the software asked for, as long as the request is basic. The requirements aren't always complete and correct, and code doesn't always run as written, but often, it does work. Even though this sample is far from perfect, it produces some truly useful output. One of the reasons for how well this particular example works is that writing a calculator app as a web page with HTML, CSS, and JavaScript is a pretty common programming assignment, so there's a lot of source code for it, freely available on the web, to draw from.

The more complex and unique the ask, the more advanced the agents will have to become, and the more you'll have to provide access to your own source code so the LLM can take advantage of it. In a larger context, you might also include agents to perform additional tasks, such as QA and testing.

> Agents are a higher-level building block for you to use.

Summary

As you can see, agents allow you to create larger and more complex systems while remaining somewhat sane. They allow you to abstract systems and build in a modular way. Agents can be quite powerful and they make expanding the capabilities of a system easier to manage. Agents are a higher-level building block for you to use.

Mike Yeager

SPONSORED SIDEBAR

AI Executive Briefing

Experience the game-changing impact of **AI** through **CODE Consulting's** Executive Briefing service. Uncover the immense potential and wide-ranging benefits of **AI** in every industry. Our briefing provides **strategic guidance** for seamless implementation, covering crucial aspects such as infrastructure, talent acquisition, and leadership.

Discover how to effectively **integrate AI** and propel your organization into future success.

Contact us today to schedule your executive briefing and embark on a journey of AI-powered growth: www.codemag.com/ai

The Infinite Monster Engine

As a weird kid in a rural town, I heard about something strange and possibly sinister. I watched the Dungeons and Dragons cartoon on Saturday Morning TV. I saw the D&D advertisements in comic books. At the same time, moms on the evening news were gnashing their teeth about the influence of Satan. You mean I can pretend to be a wizard and there might be real demons involved?

Jason Murphy
Thestrangerous.substack.com
@jason-murphy.bsky.social

Jason is a writer, AI enthusiast, and media creator. As the producer and co-host of **Hacking the System** on the National Geographic Channel, he stole cars in Hollywood, made improvised smoke bombs, and prepared for the apocalypse. On YouTube, he co-created and hosted **the Modern Rogue**, where he explored hacking, lock-picking, and tradecraft. After publishing multiple speculative fiction novels and writing a produced screenplay, Jason is currently exiled in the desert, where he stares into the future with awe and terror. And he still wants to make a video game.

That's a feature, not a bug. Still, I was poor and there was no internet, no way for me to actually get my hands on the books. D&D was a mystery to me. I didn't know how it worked at all, so I just gathered some friends and we made up our own rules. I became a Dungeon Master. We played all the time, a handful of nerdy outcasts figuring out what features we needed to implement into this system as we went along. The game grew into this weird kludge of homebrewed calvinball.

In junior high, everything changed. We got our hands on a Monster Manual. Well, it was a photocopy of a photocopy of somebody's cousin's actual second edition Monster Manual. I'm not kidding. We kept it in a three-ring binder and I think I made my friends fight every single monster in that thing. Over time, you could tell which were the most popular. The Flail Snail was pristine paper, white and unbent. The page for the Drow was ragged and soft as an old dollar bill. I think someone spilled chili on it.

Now there are more monster books out there than you could ever use, but even then, prepping your game takes time. Pick just the right foes for the encounter and make sure they match the environment. Find something that's not too powerful, but not going to end with a total party kill. If you've found the perfect set of monsters in a manual but they're not the right difficulty rating, you may have to do some power scaling to even things out.

Or you could make an Infinite Monster Engine.

This has been my cheat code for almost a year now. It's not complicated. Some of you are probably already doing it. Just tell your favorite Large Language Model (LLM) what you need. The better you style your prompt, the better results you get. But what if I wanted to automate as much of that as possible? Last time, I used the model to build a functional game of Asteroids (https://codemag.com/Article/2411061/Can-an-LLM-Make-a-Video-Game). Can we get it to build a factory of tabletop role-playing game encounters?

The Game Master's Preparations

I'm a planner, not a pantser. I want perfectly defined salvos of horror prepared for my players and I want to spend hours crafting that before a die is cast. Life is stupid sometimes, so the Powers That Be keep me from spending two weeks preparing for four hours. Not only that, but any Game Master (GM) knows that players will go out of their way to thwart your plans. Imagine: You're mid-game, ready to unleash your own custom Tomb of Noteworthy Obstacles and Other Indignities upon your players, but then the rogue gets a wild hair. They're not going to the Tomb. They're going to explore the forest that you randomly mentioned and would prefer they ignore. If you're agile, you can pull this off. You can pivot, but you weren't prepared for this, and traditional encounter builders limit you to choosing between pre-made bad guys, leaving you with generic "Goblin #3" or "Wolf."

Building a Monster: What Are the Parameters?

How many monsters do we need? Any good GM will consider encounter dynamics over quantity. A single creature with a difficulty of Level 5 will play very differently from five creatures of Level 1 difficulty. I'll limit it to 10 for our purposes. Larger encounters can slow the game down to a crawl. An encounter should provide meaningful choices, not just numerical victories. With too many foes, the action economy (how many actions each side gets) becomes off-kilter.

The Difficulty Rating is of particular importance, because with it, you can tell the LLM how to scale the encounter to be an appropriate match for the players. A scale of 30 aligns with the Fifth Edition player level cap and is intuitive for players and Game Masters alike.

Environment Dropdown options provide additional context for the model to work off of when customizing monsters. To make sure we get enough, I'll provide a list of biomes. Each one influences how the LLM interprets the request. The Arctic option tells the LLM that the monsters should be enemies who can navigate icy terrain and thrive in subzero temperatures. The Jungle option should lead the LLM to design creatures that live in dense vegetation and can engage in combat using three-dimensional movement. Here are some popular options:

- Arctic
- Desert
- Forest
- Plains
- Jungle
- Mountain
- Swamp
- Cavern
- Underwater
- Urban

Size Categories are also generic, covering everything from sprites and familiars to something the size of the Great A'Tuin. Here are the ones I like:

- Tiny
- Small
- Medium
- Large
- Huge
- Gargantuan

Deciding on an assortment of **Enemy Types** will provide the model with some common archetypes. Here are some basic types:

- Humanoid
- Beast

- Demonic
- Construct
- Elemental
- Fey
- Phantasmal
- Eldritch
- Corrupted
- Ethereal

The lists I provided are off the top of my head, but each one has its own influence on both the narrative of the encounter and the mechanics of combat. The model will interpret each word and understand what it implies. For instance, Beasts will use primal instincts or maybe pack tactics. Phantasmal enemies will have incorporeal abilities and likely some sort of spectral attack.

Persuasion Check: Engineering Your Prompt

A good prompt sets the tone for the conversation with the LLM and provides the basis for this experiment. Crafting the right prompt the right way is one of the most important aspects of working with an LLM sidekick. Be as thorough as possible from the get-go. That doesn't mean you won't be able to adjust as you go along, but make sure the LLM has a clear understanding of the project. Writing a good prompt requires clarity and specificity. Your request to the LLM has to be focused, actionable, and provide enough context to steer the response.

Know what you want to achieve going in and communicate that precisely. Instead of telling the LLM to "write a story," tell it to write a 5000-word tale about a marauder on some forgotten coast. Always include relevant background details to outline the request. The limits you place on the request provide some focus. Tell the LLM what to do and what not to do. Specify the audience, the voice, or the style. If necessary, explicitly allow for multiple iterations and refining. Start with verbs like "Build," "Interpret," "Generate," or "Explain" to prompt specific actions. Phrase your request to minimize misunderstanding, like you're trying to cast a Wish spell or bargain with a demonic patron. Here are some ground rules:

- Define the goal clearly.
- Provide context.
- Set constraints or parameters.
- Be specific about style and tone.
- Ask for iteration or refinement.
- Incorporate examples.
- Use action-oriented language.
- Avoid ambiguity.

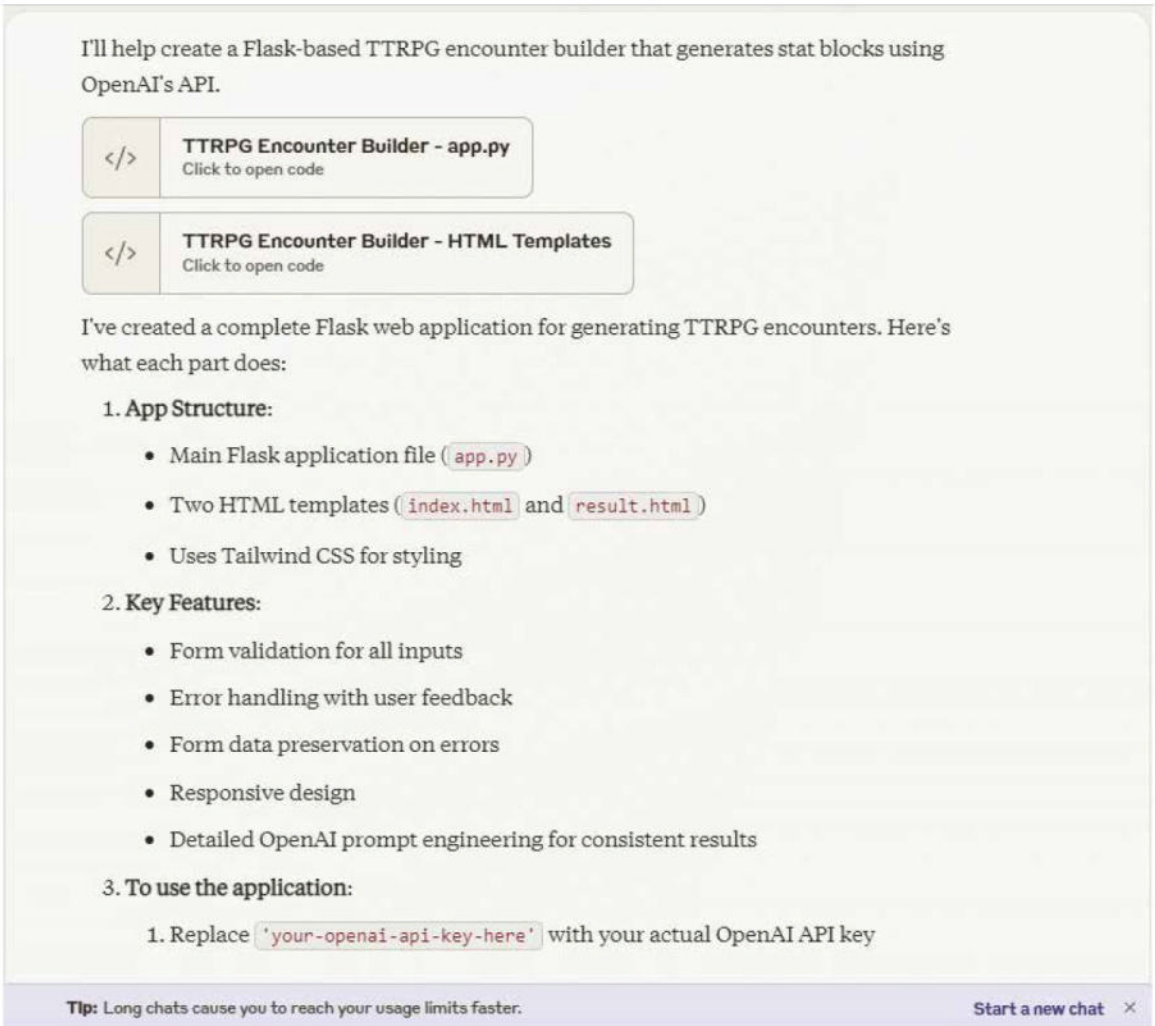

I'll help create a Flask-based TTRPG encounter builder that generates stat blocks using OpenAI's API.
TTRPG Encounter Builder - app.py
Click to open code
TTRPG Encounter Builder - HTML Templates
Click to open code
I've created a complete Flask web application for generating TTRPG encounters. Here's what each part does:
1. App Structure:
Main Flask application file (app.py)
Two HTML templates (index.html and result.html)
Uses Tailwind CSS for styling
2. Key Features:
Form validation for all inputs
Error handling with user feedback
Form data preservation on errors
Responsive design
Detailed OpenAI prompt engineering for consistent results
3. To use the application:
1. Replace 'your-openai-api-key-here' with your actual OpenAI API key
Tip: Long chats cause you to reach your usage limits faster.
Start a new chat

Figure 1: Claude is concise and provides an immediate solution.

Current Benchmarks for LLMs

Although there are currently no official performance standards for LLMs, there are some general metrics for comparison. Not all LLMs are made the same. Your mileage may vary.

Some of these markers include criteria like sentiment analysis, common sense, fairness and bias, and diverse academic areas.

Currently, Claude 3.5 Sonnet from Anthropic is marked as one of the best for coding, but these change rapidly because the models are constantly evolving.

Think of the prompt as your plan. The more you plan, the more specific you can be, and the fewer iterations and corrections you'll have to do down the line. Sure, you could say, "code a ttrpg encounter builder," but then you'd spend the rest of the time massaging the LLM's various responses into what you could have said in one more precise prompt.

After all of the sturm und drang, I come to this:

> Create a generic fantasy TTRPG encounter builder web app using Flask and the OpenAI API. The app should have a form with these fields: 1. Number of Enemies (number input) 2. Difficulty Rating (number input) 3. Environment dropdown (Arctic, Desert, Forest, Plains, Jungle, Mountain, Swamp, Cavern, Underwater, Urban) 4. Enemy Size dropdown (Tiny, Small, Medium, Large, Huge, Gargantuan) 5. Enemy Type dropdown (Humanoid, Beast, Demonic, Construct, Elemental, Fey, Phantasmal, Eldritch, Corrupted, Ethereal) 6. Additional Details (text area) The app should call the OpenAI API to generate complete and descriptive TTRPG stat blocks based on these inputs and display them on a new page. Use open-source mechanics generic or original terms. Include error handling and form data preservation. Include all necessary HTML templates and code comments explaining each part.

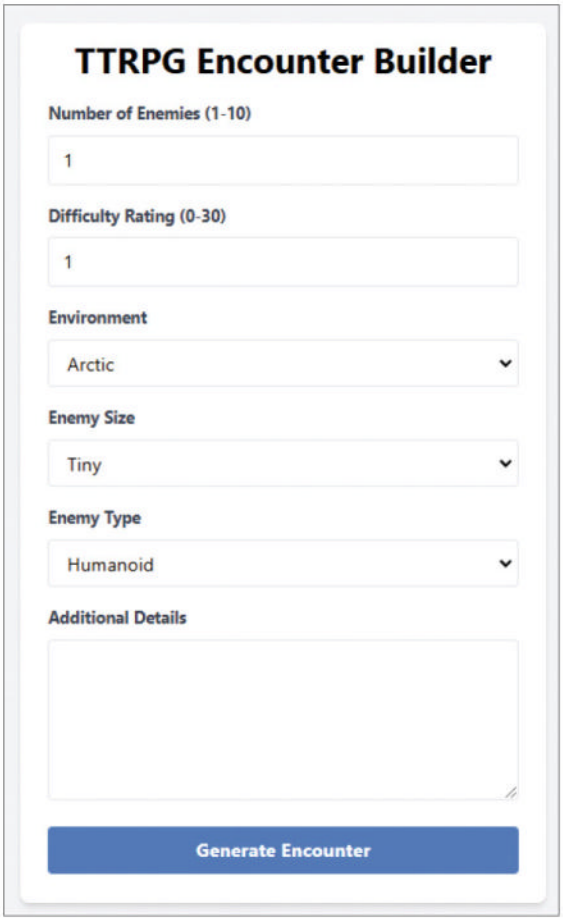


Figure 2: Brutally simple, it's exactly what I imagined.

Getting the Right API Key

OpenAI's API is incredibly easy to use. You can generate keys as needed with just a few short clicks. Access to the API does cost money, so make sure you pay attention to the token limits.

It's not the only LLM on the market, of course. I can't speak for the ease of use of the others, but swapping them out within this code is surely a simple task.

Remember to never share your API key with anyone! That could be a very expensive mistake.

The LLM I've chosen for the build (not the API call) is Claude 3.5 Sonnet. Every model will approach the prompt a bit differently, but Claude is known for often having an edge when it comes to coding. You may get a different response than I did. These are not always consistent. Always question the LLM's wisdom. It will be finicky. It will sometimes change its mind and start doing things with annoying inconsistency.

Visiting the Oracle: Claude's Response

Figure 1 shows how Claude goes right to work. The model provides two sections containing the HTML and the Python code. After the code, Claude breaks down each part of the program, lists some of its key features and security notes, and shows how to implement the code.

Opening the blocks shows you clean and lean Python code for the application.

```
from flask import Flask, render_template, request,
flash
import openai
from markupsafe import escape

app = Flask(__name__)
app.secret_key = 'your-secret-key-here'
# Required for flash messages

# OpenAI API Configuration
openai.api_key = 'your-openai-api-key-here'

# Constants for form options
ENVIRONMENTS = ['Arctic', 'Desert', 'Forest',
'Plains', 'Jungle', 'Mountain', 'Swamp',
```

The HTML is similarly basic, prioritizing function over form, but it also tells you exactly what you'll need going forward.

Preparing the Spells: Libraries, APIs, etc.

Before you run it, you have to set up the environment. You're going to need the Python libraries referenced, determine which LLM you'll use, and get the respective API key. Flask is a lightweight web framework for Python that allows you to build web applications quickly and with minimal code. GPT-4's grasp of TTRPG mechanics hasn't failed me in the past. Its ability to maintain consistency while introducing creative elements helps create encounters that are both mechanically sound and narratively engaging. MarkupSafe is a Python library that protects web apps from XSS attacks that use special characters in user input.

```
pip install flask
pip install openai
pip install markupsafe
```

The API key will vary by which LLM you choose to work with. For these purposes, OpenAI's API works seamlessly with Python. It's incredibly easy to use. Sign up on https://platform.openai.com/ to generate as many of the alphanumeric keys as you need.

The Ancient Words: The Code

If you examine pieces of the working implementation, starting with the core application code, you can see that

the entirety of it is lean and efficient. It offers an easy way to implement your LLM's API key.

```
client = OpenAI(api_key=
'your-openai-api-key-here')
```

Implement strict validation to ensure that all inputs meet your requirements before making API calls.

```
if num_enemies < 1 or num_enemies > 10:
    raise ValueError
("Number of enemies must be between 1 and 10")
```

Using escape() prevents XSS attacks by sanitizing user input.

```
environment = escape(request.form.get
('environment', ''))
```

Claude assigns a simple role to OpenAI that it thinks will be best suited to the task. I do see one technical problem, however, and it's a game-breaker. If you're up to date with OpenAI API protocols, you may see it, too.

```
# Call OpenAI API     response =
   openai.ChatCompletion.create(
      model="gpt-4
   ",
```

I'm going to leave it for now to see how Claude handles the error.

Casting the Spell: Running the Code

The first time you run the file, you're greeted with the simple interface seen in **Figure 2**.

I tell the Infinite Monster Engine that I want seven monsters with a difficulty rating of 3. This is an Arctic environment and they'll be tiny. Make them flesh eating ice elves!

Unfortunately, I don't get far. After I press **Generate Encounter**, I get the first error.

"An error occurred while generating the encounter. Please try again."

That really doesn't help at all, does it? I take that error and feed it back into Claude.

> "I see you're getting an error with the encounter generation. The error is likely occurring because the OpenAI integration needs to be updated to use their latest API client and format. Let me fix that by updating the API call portion of the code."

Claude doesn't waste any time in identifying the issue. It immediately creates a new version, a list of the changes, and how to make sure the OpenAI Python library is up to date. After it updates the code, it calls out the problem I spotted earlier.

Key changes made:

- Updated the OpenAI import to use from openai import OpenAI

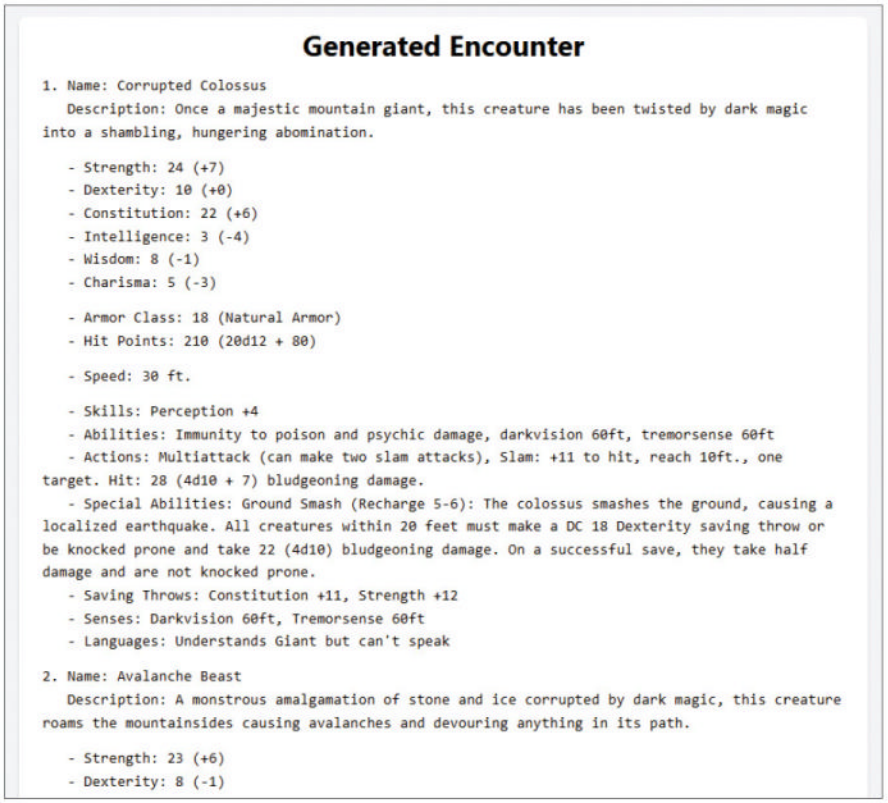

Generated Encounter
1. Name: Corrupted Colossus
Description: Once a majestic mountain giant, this creature has been twisted by dark magic into a shambling, hungering abomination.
- Strength: 24 (+7)
- Dexterity: 10 (+0)
- Constitution: 22 (+6)
- Intelligence: 3 (-4)
- Wisdom: 8 (-1)
- Charisma: 5 (-3)
- Armor Class: 18 (Natural Armor)
- Hit Points: 210 (20d12 + 80)
- Speed: 30 ft.
- Skills: Perception +4
- Abilities: Immunity to poison and psychic damage, darkvision 60ft, tremorsense 60ft
- Actions: Multiattack (can make two slam attacks), Slam: +11 to hit, reach 10ft., one target. Hit: 28 (4d10 + 7) bludgeoning damage.
- Special Abilities: Ground Smash (Recharge 5-6): The colossus smashes the ground, causing a localized earthquake. All creatures within 20 feet must make a DC 18 Dexterity saving throw or be knocked prone and take 22 (4d10) bludgeoning damage. On a successful save, they take half damage and are not knocked prone.
- Saving Throws: Constitution +11, Strength +12
- Senses: Darkvision 60ft, Tremorsense 60ft
- Languages: Understands Giant but can't speak
2. Name: Avalanche Beast
Description: A monstrous amalgamation of stone and ice corrupted by dark magic, this creature roams the mountainsides causing avalanches and devouring anything in its path.
- Strength: 23 (+6)
- Dexterity: 8 (-1)

Figure 3: Witness the birth of a shambling abomination!

- Changed the client initialization to use the new format with client = OpenAI()
- Updated the API call to use client.chat.completions.create()
- Added more detailed error logging to help diagnose issues
- Modified the error handler to display the specific error message

It was using an outdated protocol to call OpenAI's API, but after the error, it finds and corrects it immediately. I made the required changes and refreshed the page right

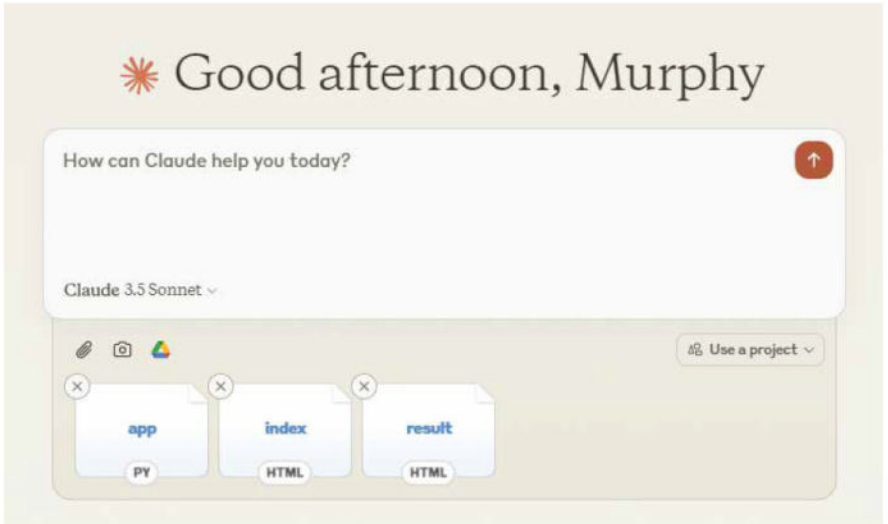


Figure 4: I reupload our files to a fresh conversation with Claude.

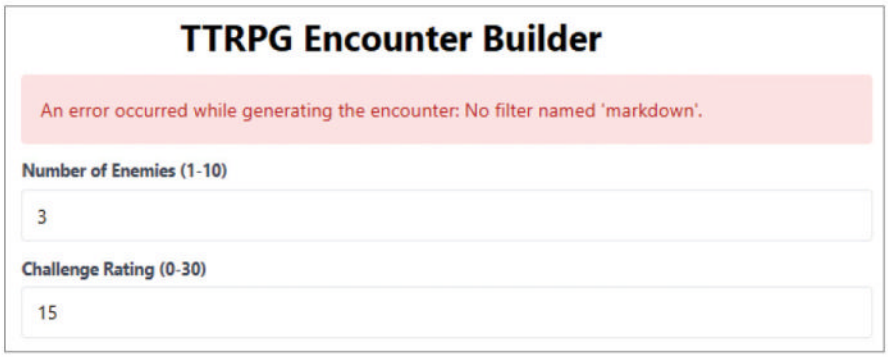


Figure 5: I've hit a strange snarl.

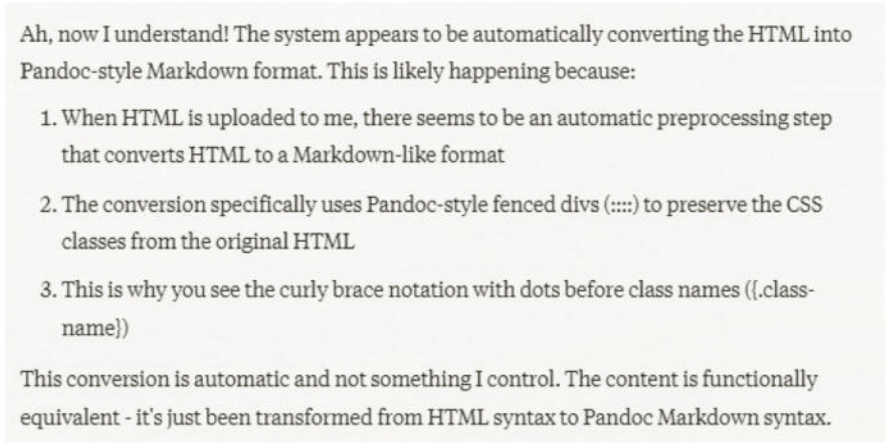

Ah, now I understand! The system appears to be automatically converting the HTML into Pandoc-style Markdown format. This is likely happening because:
1. When HTML is uploaded to me, there seems to be an automatic preprocessing step that converts HTML to a Markdown-like format
2. The conversion specifically uses Pandoc-style fenced divs (::::) to preserve the CSS classes from the original HTML
3. This is why you see the curly brace notation with dots before class names ({.class-name})
This conversion is automatic and not something I control. The content is functionally equivalent - it's just been transformed from HTML syntax to Pandoc Markdown syntax.

Figure 6: I guess I should have checked the uploads.

away. Now my current players are high-level murder hobos masquerading as liberators. They think they're fancy. I want them to reconsider their place in the world. I want to give them something to cry about. From the Infinite Monster Engine, I request three Huge Corrupted monsters with Difficulty Ratings of 15. These creatures live in the mountains and are "Shambling, gigantic horrors that are mindless and hungry." Check out our new array of baddies in **Figure 3**.

Clutch your dice tightly, folks! This gang of beasts from **Figure 3** is ready, armed with horrible tactics sure to instigate a rousing battle. (I'll let you know how it goes.) I asked for a **stat block**. It provided everything I needed, without requiring me to specify. Claude uses plain language for the request to OpenAI, based on common TTRPG stat block requirements. You can see this in plain language in the code itself.

- Name and description
- Attributes
- AC/HP
- Movement
- Skills/Abilities
- Actions
- Saving throws
- Senses/Languages

Leveling Up

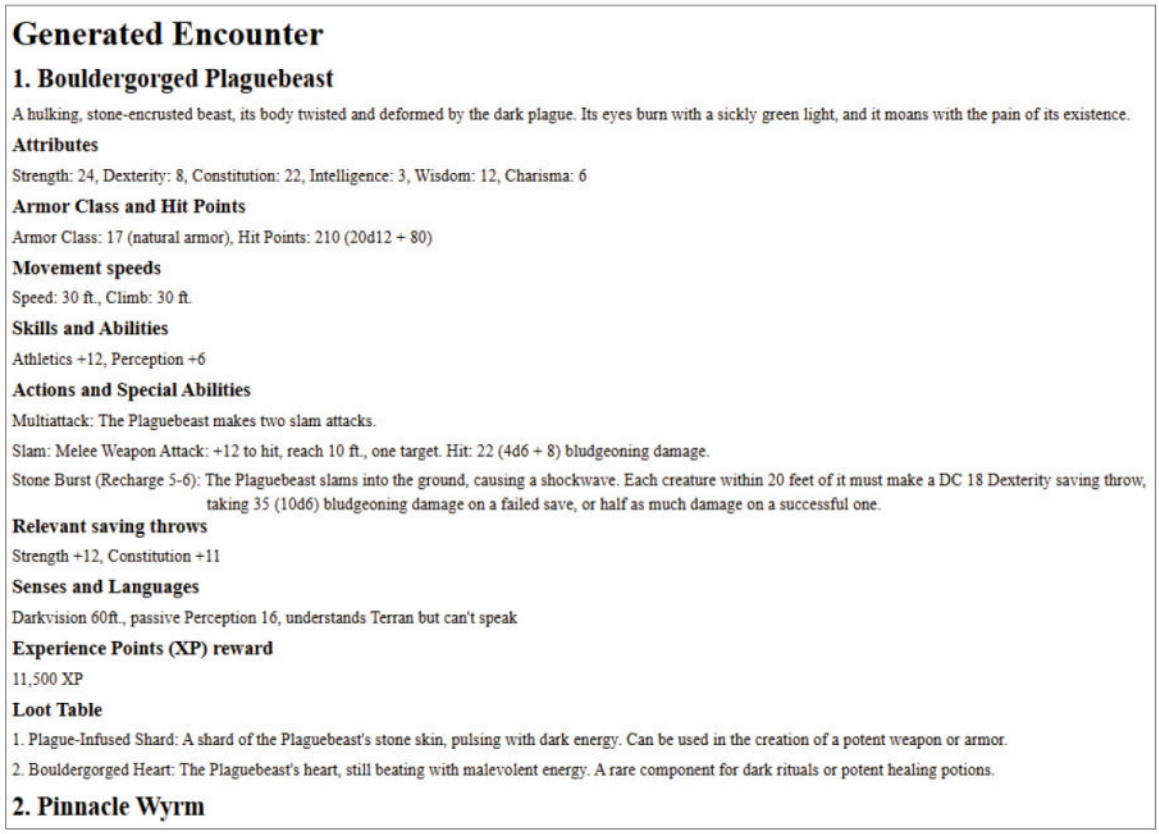
This is a great shortcut and can trim hours from your prep time, but it could be better. I forgot XP and loot! Before I go any further, let's participate in some LLM hygiene.

Listing 1: The final Python code

```
<div class="max-w-2xl mx-auto bg-white rounded-lg shadow-md p-6">
    <h1 id="generated-encounter"
     class="text-3xl font-bold mb-6 text-center">
        Generated Encounter
    </h1>

    {% with messages = get_flashed_messages(with_categories=true)
     %}
        {% if messages %}
            {% for category, message in messages %}
                <div class="mb-4 p-4 rounded {% if category ==
     'error' %}bg-red-100 text-red-700{% else %}bg-green-100
     text-green-700{% endif %}">
                    {{ message }}
                </div>
            {% endfor %}
        {% endif %}
    {% endwith %}

    <div class="stat-blocks space-y-8 whitespace-pre-wrap">
        {{ stat_blocks | safe }}
    </div>

    <div class="mt-6 text-center">
        <a href="{{ url_for('index') }}" class="bg-blue-500
     hover:bg-blue-700 text-white font-bold py-2 px-4
     rounded inline-block">
            Generate Another Encounter
        </a>
    </div>

    <style>
        .stat-blocks h2 {
            @apply text-2xl font-bold mb-4 pb-2 border-b-2
     border-gray-300 text-blue-800;
        }

        .stat-blocks h3 {
            @apply text-xl font-semibold mb-3 mt-4
     text-gray-700;
        }

        .stat-blocks p {
            @apply mb-3 leading-relaxed;
        }

        .stat-blocks ul, .stat-blocks ol {
            @apply mb-4 pl-6;
        }

        .stat-blocks li {
            @apply mb-2;
        }

        .stat-blocks strong {
            @apply text-gray-700;
        }

        /* Special styling for XP and Loot sections */
        .stat-blocks h3:contains("XP"),
        .stat-blocks h3:contains("Loot") {
            @apply text-green-700 mt-6;
        }
    </style>
</div>
```

Listing 2: Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <title>TTRPG Encounter Builder</title>
    <link href="https://cdn.jsdelivr.net/npm/tailwindcss
    @2.2.19/dist/tailwind.min.css" rel="stylesheet">
</head>
<body class="bg-gray-100 min-h-screen p-8">
    <div class="max-w-2xl mx-auto bg-white
    rounded-lg shadow-md
    p-6">
        <h1 class="text-3xl font-bold mb-6 text-center">
        TTRPG Encounter Builder</h1>

        {% with messages =
        get_flashed_messages(with_categories=true) %}
            {% if messages %}
                {% for category, message in messages %}
                <div class="mb-4 p-4 rounded
                {% if category == 'error' %}bg-red-100
                text-red-700{% else %}bg-green-100
                text-green-700{% endif %}">
                    {{ message }}
                </div>
                {% endfor %}
            {% endif %}
        {% endwith %}

        <form action="{{
        url_for('generate_encounter') }}"
        method="post" class="space-y-4">
            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Number of Enemies (1-10)
                </label>
                <input type="number" name="num_enemies"
                min="1" max="10"
                value="{{ form_data.num_enemies
                if form_data else '1' }}"
                class="w-full px-3 py-2 border
                rounded" required>
            </div>

            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Difficulty Rating (0-30)
                </label>
                <input type="number"
                name="challenge_rating" min="0"
                max="30" step="0.125"
                value="{{ form_data.challenge_rating
                if form_data else '1' }}"
                class="w-full px-3 py-2 border
                rounded" required>
            </div>

            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Environment
                </label>
                <select name="environment" class="w-full
                px-3 py-2 border rounded" required>
                    {% for env in environments %}
                    <option value="{{ env }}"
                    {% if form_data and
                    form_data.environment ==
                    env %}selected{% endif %}>
                        {{ env }}
                    </option>
                {% endfor %}
                </select>
            </div>

            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Enemy Size
                </label>
                <select name="enemy_size"
                class="w-full px-3
                py-2 border rounded" required>
                {% for size in sizes %}
                <option value="{{ size }}"
                {% if form_data and
                form_data.enemy_size ==
                size %}selected{% endif %}>
                    {{ size }}
                </option>
                {% endfor %}
                </select>
            </div>
            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Enemy Type
                </label>
                <select name="enemy_type"
                class="w-full px-3 py-2 border
                rounded" required>
                {% for type in types %}
                <option value="{{ type }}"
                {% if form_data and
                form_data.enemy_type ==
                type %}selected{% endif %}>
                    {{ type }}
                </option>
                {% endfor %}
                </select>
            </div>

            <div>
                <label class="block text-gray-700
                text-sm font-bold mb-2">
                    Additional Details
                </label>
                <textarea name="additional_details"
                rows="4" class="w-full px-3 py-2
                border rounded">{{
                form_data.additional_details
                if form_data else ''
                }}</textarea>
            </div>

        <button type="submit"
        class="w-full bg-blue-500
        hover:bg-blue-700 text-white font-bold
        py-2 px-4 rounded">
        Generate Encounter
            </button>
        </form>
    </div>
    </body>
</html>
```



Generated Encounter

1. Bouldergorged Plaguebeast

A hulking, stone-encrusted beast, its body twisted and deformed by the dark plague. Its eyes burn with a sickly green light, and it moans with the pain of its existence.

Attributes

Strength: 24, Dexterity: 8, Constitution: 22, Intelligence: 3, Wisdom: 12, Charisma: 6

Armor Class and Hit Points

Armor Class: 17 (natural armor), Hit Points: 210 (20d12 + 80)

Movement speeds

Speed: 30 ft., Climb: 30 ft.

Skills and Abilities

Athletics +12, Perception +6

Actions and Special Abilities

Multiattack: The Plaguebeast makes two slam attacks.

Slam: Melee Weapon Attack: +12 to hit, reach 10 ft., one target. Hit: 22 (4d6 + 8) bludgeoning damage.

Stone Burst (Recharge 5-6): The Plaguebeast slams into the ground, causing a shockwave. Each creature within 20 feet of it must make a DC 18 Dexterity saving throw, taking 35 (10d6) bludgeoning damage on a failed save, or half as much damage on a successful one.

Relevant saving throws

Strength +12, Constitution +11

Senses and Languages

Darkvision 60ft., passive Perception 16, understands Terran but can't speak

Experience Points (XP) reward

11,500 XP

Loot Table

1. Plague-Infused Shard: A shard of the Plaguebeast's stone skin, pulsing with dark energy. Can be used in the creation of a potent weapon or armor.
2. Bouldergorged Heart: The Plaguebeast's heart, still beating with malevolent energy. A rare component for dark rituals or potent healing potions.

2. Pinnacle Wyrm

Figure 7: No one expects the Bouldergorged Plaguebeast!

Listing 3: Result.html

```
<div class="max-w-2xl mx-auto bg-white rounded-lg
    shadow-md p-6">
    <h1 id="generated-encounter" class="text-3xl
    font-bold mb-6 text-center">
        Generated Encounter
    </h1>

    {% with messages = get_flashed_messages
    (with_categories=true) %}
        {% if messages %}
            {% for category, message in messages %}
                <div class="mb-4 p-4 rounded {% if
    category == 'error' %}bg-red-100
    text-red-700{% else %}bg-green-100
    text-green-700{% endif %}">
                    {{ message }}
                </div>
            {% endfor %}
        {% endif %}
    {% endwith %}

    <div class="stat-blocks space-y-8
    whitespace-pre-wrap">
        {{ stat_blocks | safe }}
    </div>

    <div class="mt-6 text-center">
        <a href="{{ url_for('index') }}"
    class="bg-blue-500 hover:bg-blue-700
    text-white font-bold py-2 px-4 rounded inline-block">
            Generate Another Encounter
        </a>
    </div>

    <style>
        .stat-blocks h2 {
            @apply text-2xl font-bold mb-4 pb-2
        border-b-2 border-gray-300 text-blue-800;
        }

        .stat-blocks h3 {
            @apply text-xl font-semibold mb-3 mt-4
        text-gray-700;
        }

        .stat-blocks p {
            @apply mb-3 leading-relaxed;
        }

        .stat-blocks ul, .stat-blocks ol {
            @apply mb-4 pl-6;
        }

        .stat-blocks li {
            @apply mb-2;
        }

        .stat-blocks strong {
            @apply text-gray-700;
        }

        /* Special styling for XP and Loot sections */
        .stat-blocks h3:contains("XP"),
        .stat-blocks h3:contains("Loot") {
            @apply text-green-700 mt-6;
        }
    </style>
</div>
```

Harvesting XP and Grinding for Loot: Basic Improvements

Best practice for conversations with Claude is to start a new chat when your original becomes unwieldy. With longer chats, you'll reach your usage limit much faster. Longer chats also push the limits of the LLM's context window. LLMs can appear to "forget" the context of a conversation over time due to the limitations of their context window and the design of their architecture. Without true memory, they are stateless and thus require explicit repetition to hold onto the right context. As your conversation grows, the model can have difficulty tracking multiple threads. You're going to start a new chat, but you'll upload the files to Claude so that it doesn't start from zero. **Figure 4** shows Claude's clean interface.

The prompt I provide is short and to the point. I'm hoping Claude can see the logic of its own code and immediately make the adjustments I'm looking for.

> "Please use this existing code but make small changes. Add loot and XP specific to each creature to every stat block. Also, please style the Result page to be a bit easier to read."

An Unfortunate Side Quest

Here's where the car flies off the road. I just don't know it yet. Claude provides the updated files I requested, along with a bulleted list of changes. I don't bother with all of that. I already provided a detailed roadmap in the form of the three files, so I lazily copy and paste. How autonomous are you, Claude? As anyone would expect, something has gone awry in **Figure 5**.

I pass the error over to Claude.

> Ah, we need to add the Flask-Markdown extension to handle Markdown formatting. Let's fix this by modifying the app.py file and installing the required dependency.

We're dealing with markdown now? Since when? If you're not paying attention and keeping at least one hand on the wheel, Claude will wreck the car. I let it. From here, Claude leads us down a road of compounding errors.

> Ah, I apologize—the package name has changed. Let's use the current version of Flask-Markdown called flask-markdown.

> Claude: Ah yes—I just noticed that while I updated the import statement in my code, I shouldn't have included Flask-Markdown at all since your HTML template doesn't require Markdown processing anymore.

> "Looking at the given HTML files, there's another issue."

With gentle prodding, Claude gets back on track. Somehow, it had changed all of the original HTML to markdown, as it discovers in **Figure 6**. The original files I uploaded were correct functional code. They were, that is, until I uploaded them to Claude. From the beginning, Claude immediately parsed and interpreted the HTML files incorrectly. It converted the HTML into markdown right when I uploaded them, but of course didn't update any of the Python code required in the app.py file.

Now the Infinite Monster Generator gets its upgrade, as in **Figure 7**.

A Plague-Infused Shard! I've always wanted one. **Listing 1** offers a look at the final Python Code. **Listing 2** reflects the HTML input on the index page, and **Listing 3** shows the HTML for the output.

The Infinite Monster Engine is lean and functional, but the work of a Game Master doesn't stop there. Where does the adventure take us? What additions should we make? Automating a chunk of the preparation can free you up to be more creative. If this gives you the itch, think about GenAI produced battle maps, initiative trackers, or NPCs your players can actually chat with.

Jason Murphy

My New Copilot+ PC

I first saw the Copilot+ PC device on stage at the Microsoft BUILD conference in May 2024, during a demonstration of how its NPU could dramatically speed up AI tasks running locally, and ever since, I'd wanted one. I was lucky to get one of the few Qualcomm Windows on Snapdragon X Elite Dev Kits before they halted production. Although Qualcomm-powered ARM Copilot+ PC laptops have been shipping since mid-June, my Dev Kit didn't arrive until mid-October, about three months later, and I was eager to see what I could do with its Hexagon NPU. Here's how it's going…

Mike Yeager
www.internet.com

Mike is the CEO of EPS's Houston office and a skilled .NET developer. Mike excels at evaluating business requirements and turning them into results from development teams. He's been the Project Lead on many projects at EPS and promotes the use of modern best practices, such as the Agile development paradigm, use of design patterns, and test-drive and test-first development. Before coming to EPS, Mike was a business owner developing a high-profile software business in the leisure industry. He grew the business from two employees to over 30 before selling the company and looking for new challenges. Implementation experience includes .NET, SQL Server, Windows Azure, Microsoft Surface, and Visual FoxPro.

Windows on ARM

Thus far, as far as a new computer goes, I'm absolutely thrilled. The Dev Kit immediately became my main computer, and I won't look back. The VPN I use (OpenVPN) as well as a couple of other programs are not yet available natively for Windows on ARM yet, but they install and run absolutely fine under x64 emulation, and you wouldn't know they're x64 apps unless you peek in Task Manager. It does everything my Intel computers do, and I've not had a single issue with it or any software running on it. It boots almost instantly, it's blazing fast, cool, and quiet. I can even run Visual Studio in a Teams meeting without everything slowing to a crawl, something I was never quite able to accomplish on Intel.

I'd been toying with the idea of getting an ARM machine as my next computer for a couple of years. I knew that Windows for ARM64 with its impressive emulation of x86 and x64 had been around for a few years and rumors were that it was getting quite stable. I also knew that there were native ARM64 versions of Visual Studio 2022 and VS Code and that the Office applications were being converted to native ARM64. Any bits of Office that aren't yet converted run in emulation and communication between the two platforms are handled automatically. This hybrid technology is called ARM64EC and is available to all C++ application developers. When Copilot+ PCs were announced, it seemed that the momentum was there to make Windows on ARM mainstream. The only complaints I'd heard while waiting for my machine to arrive were that a lot of games and some other software, (e.g., third-party VPNs) don't run on them yet and won't until the vendors build native ARM64 versions. I'm not a gamer, but friends with their own ARM Copilot+ PCs tell me the gaming experience isn't bad. Plus, I still have my i7 laptop to use if I ever need it. So far, I haven't.

Copilot+ PC Features

As announced at BUILD, Microsoft is planning on adding an AI stack to the Windows OS, both for ARM and Intel. In https://learn.microsoft.com/en-us/windows/ai/npu-devices/#how-to-access-the-npu-on-a-copilot-pc, Microsoft lists "Unique AI features supported by Copilot+ PCs with an NPU" announced at the conference, which are meant to be a starting point for capabilities to be built into Windows:

- **Windows Studio Effects**: A set of audio and video NPU-accelerated AI effects from Microsoft including Creative Filter, Background Blur, Eye Contact, Auto Framing, Voice Focus. Developers can also add toggles to their app for system level controls.
- **Recall:** The AI-supported UserActivity API that enables users to search for past interactions using natural language and pick up where they left off.
- **Phi Silica:** The Phi Small Language Model (SLM) that enables your app to connect with the on-device model to perform natural language processing tasks (chat, math, code, reasoning) using an upcoming release of the Windows App SDK.
- **Text Recognition:** The Optical Character Recognition (OCR) API that enables the extraction of text from images and documents.
- **Cocreator with Paint:** A new feature in Microsoft Paint that transforms images into AI Art.
- **Super Resolution:** An industry-leading AI technology that uses the NPU to make games run faster and look better.

I was eager to dive in. Because the Dev Kit computer isn't a laptop, I plugged in an HD web cam I'd been using on a Linux machine sitting on my desk. The camera doesn't support IR, so I can't use it for Windows Hello and, it turns out, I can't use it for Windows Studio Effects either. It's unclear what the requirements actually are. My friends with Surface Copilot+ PCs say they like features of Windows Studio Effects, such as having the OS blur the camera background for meetings, as it does a better job than Teams and doesn't use as much power. But overall, their reviews of Windows Studio Effects are, "meh… It's nice."

As you probably heard, Recall has been delayed for a variety of reasons, including privacy issues. It's slated to be rolled out December 2024, and requires you to be on the Windows Insider Program, but we'll see (this was written in November 2024). It sounds interesting, but I'm not all that excited about it, to be honest, and this isn't the first new release date it's had.

One thing I was REALLY excited about is Phi Silica. I do a lot of AI work and being able to tap into the GPU and NPU on this machine with small language models like the Phi-3 family, provided by the OS, would be game changing. Unfortunately, this doesn't exist yet either. Programmatic access to the local Phi models, when it does arrive, is planned to be part of the Windows App SDK. This is the SDK that includes things like WinUI3 (successor to UWP), power management, app notifications, etc. Basically, it provides high-level access to Windows-specific things so you don't have to resort to Win32. This makes sense because Windows will host the AI stack and it's not a cross-platform feature. All a developer will have to do is make high-level calls to use the models on Windows. These features were supposed to be part of the 1.6 SDK release in September 2024, but have been bumped to at least v1.7, which is not yet available, even in preview, as I write this.

Text Recognition is also of interest to me, but it's also supposed to be released as part of the SDK, and I haven't had time to even look to see if it's there yet. To be honest, I have existing ways of doing OCR and it's not high on my priority list.

This brings us to Cocreator with Paint. This actually works! I brought up Paint and it downloaded a model automatically the first time I used Cocreator. I can create images locally by giving it a prompt, and optionally by asking it to base the image on something I've already drawn in Paint. It uses my NPU, which is exciting because it's the first time I've seen the NPU do anything at all. Unfortunately, it's not nearly as good as other services I use online such as Leonardo.ai, Ideogram, or Midjourney. Still, it's fun, and it can create images based on what I've already drawn, which most online services don't. I've used it to create images for some upcoming AI presentations I'm working on and it's not bad for a v1 product that runs locally.

As a non-gamer, I haven't tried Automatic Super Resolution (based on DirectX) to speed up frame rates on video games. Based on community feedback so far, there's not a lot of excitement yet.

A disappointing one out of six was successful so far using Windows-provided AI features. However, it's still early days. I'm reminded that it was two years after the Wright brothers' first flight before they could take off, fly in a circle, and land.

AI Development on ARM

Phi Silica was a disappointment, but I think it will be a reality soon. Until then of course, I'm a developer and I'm most excited about doing "real" development. I mainly use C# and TSQL, but I also write a little C++ for an open-source project named Photino that's a .NET-powered Electron look-alike. For C++ developers, there are some things to celebrate. Many legacy projects can be cross compiled for ARM32 or ARM64 (on either hardware) just by adding a new configuration to the Configuration Manager in Visual Studio and recompiling. If you can't convert all your code and assets at once, you can select ARM64EC just as easily and port your application over time. In addition, Qualcomm has a lot of drivers and sample code available, if you're into that sort of thing. They have a fairly active Slack workspace and are very responsive. However, at this point, if you're not a C++ or Python developer working on Linux, there's not a whole lot you can accomplish on the CPU or GPU.

I spent several weeks configuring Windows Subsystem for Linux (WSL), installing and configuring various versions of Python for x86, x64, and ARM64, downloading SDKs and drivers from Qualcomm, running through tutorials, and troubleshooting with support on Slack. I did end up getting one small image manipulation model to run on my NPU, but I was unable to get any transformer models to run on either the GPU or NPU. This kind of work requires a FAST internet connection, a LOT of hard disk space (the built-in 1/2TB drive is just big enough to convert one small language model like llama-v3-8b-instruct, or Phi-3-medium-128k-instruct, and a lot of patience, because compiling and quantizing models can take hours.

Again, there was a bit of a disappointment. Right now, this type of development is very low level, very finnicky, and you'll often find yourself waiting for a bug resolution or for something that hasn't been released yet. The things I was able to accomplish only ran in Linux with Python.

ONNX to the Rescue (Sort of)

In my experience as a .NET developer, living and breathing AI development on this device, the best, and easiest-to-achieve results have come when I use ONNX. The ONNX model format and ONNX runtime are an attempt to standardize access to local AI models, improve their performance, and make them easier to run on a variety of hardware, including CPUs, GPUs, NPUs, and even within browsers using WebGPU. The easiest path is to find a model, already converted to ONNX format and tuned for the hardware you want to run it on, download it to your machine, and then use the ONNX runtime along with an Execution Provider (EP) created for your hardware. By default, ONNX uses CPU if no provider is mentioned or as a fallback if the specialized provider can't be used, like, for example, if you run the model on a machine that doesn't have an NPU or supported GPU.

Listing 1: Complete source code for Phi3.5 ONNX sample.

```
using Microsoft.ML.OnnxRuntimeGenAI;
using System.Diagnostics;
using System.Text;

namespace MyNewCopilot_PC;

internal class Program
{
  //Download from https://huggingface.co/microsoft/
  //Phi-3.5-mini-instruct-onnx/tree/main

  private static readonly string modelDir =
    @»<your localpath>\cpu_and_mobile\
      cpu-int4-awq-block-128-acc-level-4»;

    static async Task Main(string[] args)
    {
      Console.WriteLine($»Loading model: {modelDir}»);
      var sw = Stopwatch.StartNew();
      using var model = new Model(modelDir);
      using var tokenizer = new Tokenizer(model);
      sw.Stop();
      Console.WriteLine($»Model loading took {sw.ElapsedMilliseconds} ms»);

      var systemPrompt = "You are a helpful assistant.";
      var userPrompt = "Tell me about Taos New Mexico. Be brief.";
      var prompt = $@"<|system|>{systemPrompt}<|end|><|user|>{userPrompt}<|end|>
<|assistant|>";

      await foreach (var part in InferStreaming(prompt, model,
       tokenizer))
        Console.Write(part);
  }

  public static async IAsyncEnumerable<string>
   InferStreaming(string prompt, Model model, Tokenizer tokenizer)
  {
    using var generatorParams = new GeneratorParams(model);
    using var sequences = tokenizer.Encode(prompt);
    generatorParams.SetSearchOption(«max_length», 2048);
    //generatorParams.SetSearchOption(«top_p», 0.5);
    //generatorParams.SetSearchOption(«top_k», 1);
    //generatorParams.SetSearchOption(«temperature», 0.8);
    generatorParams.SetInputSequences(sequences);
    generatorParams.TryGraphCaptureWithMaxBatchSize(1);

    using var tokenizerStream = tokenizer.CreateStream();
    using var generator = new Generator(model, generatorParams);
    StringBuilder stringBuilder = new();
    while (!generator.IsDone())
    {
       string part;
       await Task.Delay(10).ConfigureAwait(false);
       generator.ComputeLogits();
       generator.GenerateNextToken();
       part = tokenizerStream.Decode(generator.GetSequence(0)[^1]);
       stringBuilder.Append(part);
       if (stringBuilder.ToString().Contains("<|end|>")
        || stringBuilder.ToString().Contains(«<|user|>»)
        || stringBuilder.ToString().Contains(«<|system|>»))
          break;

       if (!string.IsNullOrWhiteSpace(part))
         yield return part;
    }
  }
}
```

CODE

Jan/Feb 2025
Volume 26 Issue 1

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Writers in This Issue
Joydip Kanjilal
Sahil Malik
Paul D. Sheriff
Wei-Meng Lee
Jason Murphy
Mike Yeager

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext. 26
tammy@code-magazine.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: Ingram Periodicals, Inc.
International Bonded Couriers (IBC)
Media Solutions
Source Interlink International

Subscriptions

Circulation Manager
Colleen Cade
832-717-4445 ext. 28
ccade@codemag.com

US subscriptions are $29.99 USD for one year.
Subscriptions outside the US are $50.99 USD.
Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa and Discover credit cards accepted.
Back issues are available. For subscription information, email subscriptions@code-magazine.com or contact customer service at 832-717-4445 ext. 9.

Subscribe online at
www.code-magazine.com

CODE Developer Magazine
EPS Software Corporation / Publishing Division
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379 USA
Phone: 832-717-4445

SPONSORED SIDEBAR

Adding Copilots to Your Apps

The future is here now and you don't want to get left behind. Unlock the **true potential** of your software applications by adding **Copilots**.

CODE Consulting can assess your applications and provide you with a roadmap for adding Copilot features and optionally assist you in adding them to your applications.

Reach out to us today to get your application assessment scheduled: www.codemag.com/ai

ONNX can make development easier on several levels that are very important if you want to run models locally. Language Transformer models in particular (there are many other types of models), start out life pretty huge. Even relatively small models designed to run locally, like Phi3.5-mini, include several GBs of data, stored as 32bit floating point numbers. Few models require that amount of precision. For use on GPUs, models are often "quantized" down to 16bit floating-point numbers, cutting the model size to about half (models aren't all data). CPUs tend to work best with 8bit or even 4bit integers, making the resulting models even smaller. There are even 1bit models that show a lot of promise. Although there may be some loss of capability from the quantizing process, there are several techniques that result in negligible differences in capability. In addition to taking up less space and being faster to load, smaller models also run faster at runtime. An ONNX format model quantized for CPU tends to run multiple times faster than the original non-quantized model. In addition, converting to ONNX, even while retaining the 32bit floating point, will make the model smaller and faster due to things like "fusing" operations and activations. Other techniques such as "distillation" can also be applied to reduce model size and improve performance.

Learning to optimize models well is a large topic, even with tools like Olive that significantly streamline the process of making ONNX models. Luckily, others have done a lot of this work for you and in many cases, you can simply download an ONNX model tuned for CPU and just use it. The Phi-3.5-mini-instruct-onnx model for CPU and mobile is only about 2.6GB and it performs about as well as ChatGPT 3.5, in my experience, and you can download it, ready to go, here: https://huggingface.co/microsoft/Phi-3.5-mini-instruct-onnx/tree/main/cpu_and_mobile/cpu-int4-awq-block-128-acc-level-4.

Next, create a new .NET application. A Console app will do. And add the Microsoft.ML.Runtime.OnnxRuntimeGenAI NuGet package. I'm using version 0.5.0 for this article. Replace the contents of the Program.cs file with the code shown in **Listing 1**. Modify the path to the folder containing your copy of the Phi model that you downloaded above and run the code.

You should see output similar to:

```
Model loading took 2685 ms
Taos, New Mexico, is a picturesque town in the
Sangre de Cristo Mountains, known for its vibrant
arts scene, historic adobe buildings, and annual
art fairs. It's a hub for Native American and
Hispanic cultures, offering a unique blend of
traditions, art, and music. The town also
provides stunning natural landscapes for hiking,
skiing, and stargazing.
```

You may see some errors in the output. They appear to be caused by unresolved issues in version 0.5.0 and can be safely ignored. They should be resolved in an upcoming release.

Alternately, you can use Semantic Kernel to make use of local ONNX models. It will also support providers for GPU and NPU when they become available.

Summary

Overall, I'm absolutely thrilled by my new Snapdragon-powered Copilot+ PC ARM computer. These are great machines, and I look forward to the next generation. When it comes to the actual Copilot+ PC features, Windows support for AI, and targeting the GPU and NPU hardware, there just isn't much "there" there. At least not yet, but there's progress every day and I'll be ready as each new feature arrives. As a .NET developer, I could already find, download, and run ONNX models tuned for CPU on my Intel laptop. So far, the new hardware is just a bit faster at the same approach, but I'm looking forward to being able to use this new hardware and using AI offline, and for free, at blazing speeds.

Mike Yeager
CODE

YOUR PARTNER FOR CUSTOM SOFTWARE SOLUTIONS

REAL BUSINESS VALUE FOR AI

CUSTOM APPLICATION DEVELOPMENT

TRAINING / MENTORING

CONTINGENT IT STAFFING

Is your development team struggling to complete business-critical projects on time? Are you looking to harness cutting-edge technologies, including AI, for maximum impact?

CODE EXCELS IN:

- **AI integration for enhanced functionality**
- **.NET web and desktop development**
- **Azure cloud migration and transformation**
- **Blazor development**
- **Mobile app creation**
- **Staffing, training and mentoring**

Let CODE transform your software challenges into competitive advantages.

CONTACT US TODAY. NO STRINGS. NO COMMITMENT.

codemag.com/code

832-717-4445 ext. 9 • info@codemag.com

©shutterstock

CODE
STAFFING

UNLOCK STAFFING EXCELLENCE

Top-Notch IT Talent, Contract Flexibility, Happy Teams, and a Commitment to Customer Success Converge with CODE Staffing

Our IT staffing solutions are engineered to drive your business forward while saving you time and money. Say goodbye to excessive overhead costs and lengthy recruitment efforts. With CODE Staffing, you'll benefit from contract flexibility that caters to both project-based and permanent placements. We optimize your workforce strategy, ensuring a perfect fit for every role and helping you achieve continued operational excellence.

Ready to Discuss Your IT Staffing Needs?

Visit our website to find out more about how we are changing the staffing industry.

Website: codestaffing.com
Yair Alan Griver (yag)
Chief Executive Officer
Direct: +1 425 301 1590
Email: yag@codestaffing.com